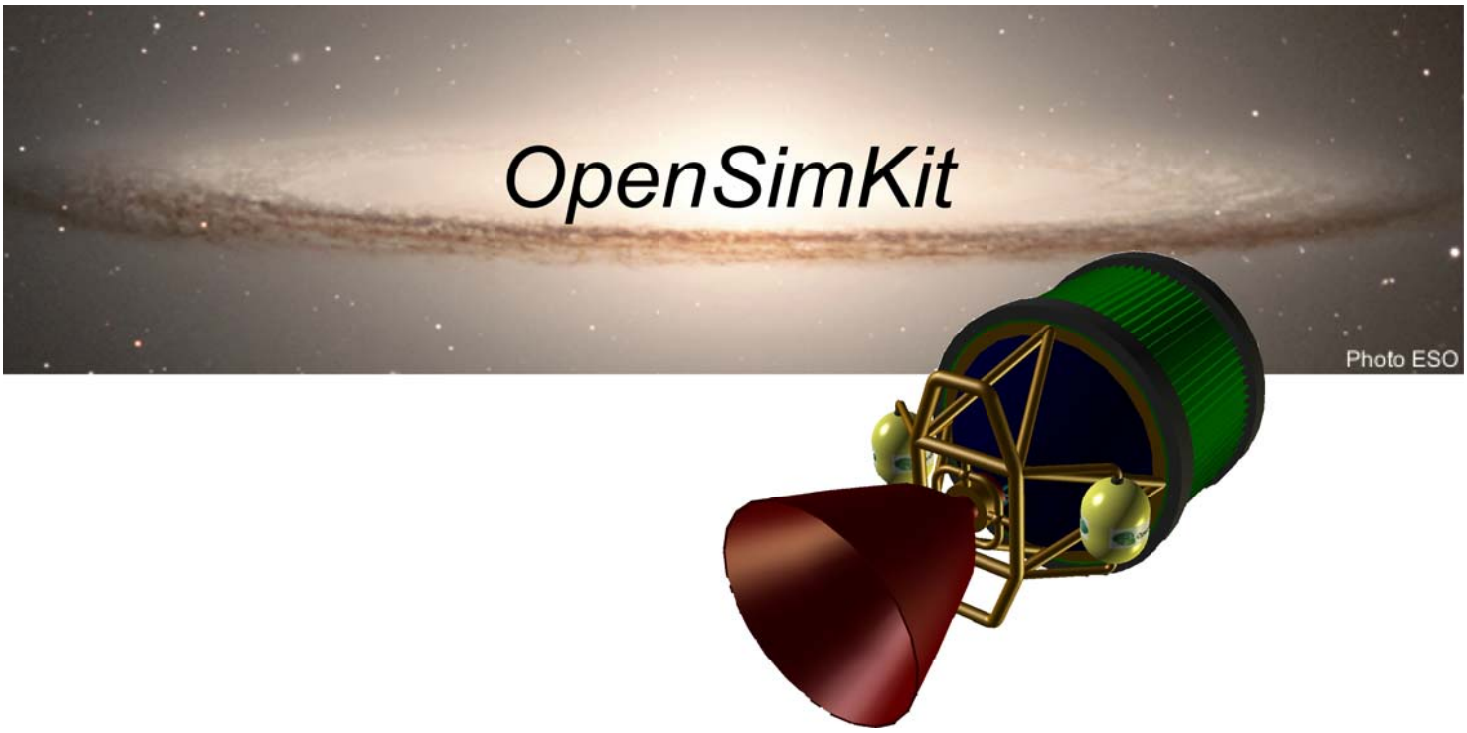



# User Manual

| | |
|---|---|
| Software Release: | 3.7.0 Java |
| Release date: | 2011-01-21 |
| Editor: | Jens Eickhoff |
| Authors: | The *OpenSimKit* Team |
| www.opensimkit.org | |

# Copyright Information

This Manual is published under Copyright of the *OpenSimKit* development Team. Distribution of the software and documentation is granted under the GNU Public License Issue 3. Please also refer to section 21 of this manual.

# Authoring team

The following team members have contributed to this manual version 3.7.0.

| | |
|---|---|
| Editor, layout, didactic concept,<br>OpenSimKit introduction,<br>tank pressurization, controllers and valve models | Jens Eickhoff |
| Infrastructure technical chapters | Alexander Brandt,<br>Jens Eickhoff |
| MMI chapter | Michael Fritz |
| Engine model chapter | Mario Kobald |
| Spacecraft structure model chapter | Michael Fritz,<br>Ivan Kossev |
| Celestia interfacing recipe | Jens Eickhoff,<br>Rouven Witt |

# Contents

1 Abbreviations....8
2 Introduction....9
3 Hardware and Operating System Prerequisites....11
4 Software Dependencies....11

## Part I

5 Downloading & Installing OpenSimKit....14
6 Building OpenSimKit Simulator and MMI Program....16
7 Quick Start Information....16
  7.1 Starting Simulation and MMI....16
  7.2 OpenSimKit MMI....17
8 Structure of an OpenSimKit XML Input File....20
  8.1 OpenSimKitConfigFile....20
  8.2 system....20
  8.3 models....21
    8.3.1 model (for model definitions)....21
    8.3.2 variable....21
    8.3.3 Model Section Example....22
  8.4 connections....23
    8.4.1 connection....23
    8.4.2 Connection Section Example....24
  8.5 providerSubscriberTable....24
    8.5.1 entry....24
    8.5.2 ProviderSubscriber Table Section Example....25
  8.6 mesh (for mesh definitions)....25
    8.6.1 meshes....25
    8.6.2 Mesh Section Example....25
  8.7 logOutput....26
    8.7.1 entry....27
    8.7.2 LogOutput Section Example....27
9 OpenSimKit Command Reference....29
  9.1 How to enter Commands....29
  9.2 Short Command Overview....29
  9.3 Extended Command Description....29
    9.3.1 Call....29
    9.3.2 Disconnect....30
    9.3.3 Get....30
    9.3.4 GetA....30
    9.3.5 Help....31
    9.3.6 Resume....32
    9.3.7 Run....32
    9.3.8 Set....32
    9.3.9 SetA....33
    9.3.10 Shutdown....34
    9.3.11 Stop....34

10 De-installing OpenSimKit....35

# Part II

11 The Rocket Stage Model Library....38
11.1 The Helium Gas Model....38
11.2 The High Pressure Bottle Model....40
11.3 The Pipe Model....41
11.3.1 Modeling the Pressure Drop....41
11.4 The Junction Model....43
11.5 The Filter Model....44
11.6 The Pressure Regulator Model....44
11.7 The Pipe Split Model....44
11.8 The Propellant Tank Model....45
11.8.1 The Differential Equation System....45
11.8.2 Thermodynamics of the Oxidizer Tank....47
11.8.3 Thermodynamics of the Fuel Tank....50
11.8.4 Thermodynamics of the Tanks in Blowdown Mode....51
11.9 The Flow Valve Model....52
11.10 The EngineController Model....52
11.11 The IntervalController Model....52
11.12 The Engine Model....53
11.12.1 Basics....53
11.12.1.1 Characteristic Velocity c* ....53
11.12.1.2 Thrust Factor cf....54
11.12.2 Model Verification....55
11.12.2.1 Verification of c* ....55
11.12.2.2 Verification of cf....56
11.13 The Structure Model....56

# Part III

12 The OpenSimKit Guide Through Galaxy....60
12.1 Introduction to Celestia....60
12.2 Concept of Interfacing Celestia from OpenSimKit....60
12.3 Steps for the Infrastructure Setup....61

# Part IV

13 Developing Component Models for OpenSimKit....68
13.1 Using Eclipse JDT 3.3.2....68
13.2 Using Netbeans 6.1....68
13.3 How to add a new Dependency to OpenSimKit....68
14 Targets of the Ant build.xml file....69
14.1 Global Targets for both the Simulator and the MMI:....69
14.2 MMI Targets:....69
14.3 Packet Library Targets:....70

14.4 Rocket Propulsion System Targets:....70
14.5 Simulator Targets:....71
15 OpenSimKit Architecture....72
15.1 Subsystem....72
15.2 Model Libraries....72
15.3 Packet Library....72
15.4 OSKPacket specification....73
16 Developing Equipment Models in OpenSimKit....75
16.1 Prerequisites....75
16.2 Writing a Java Class....75
16.3 Compiling the Java Class....78
16.4 Modifying the Input File....78
16.5 Analyzing the Output File....82
17 Coding Guidelines....83
18 Using the OpenSimKit Logger....84
18.1 Logger Code Entries for Model Classes....84
18.2 Logger Configuration for Simulation Runs....85
19 Literature....86

## Annexes

20 OpenSimKit History and Releases....90
21 License, Trademark and Warranty Disclaimer....94

# 1 Abbreviations

### General Abbreviations

| | |
|---|---|
| a.m. | above mentioned |
| cf. | confer |
| e.g. | example given |
| i.e. | Latin: id est ⇒ that is |
| w.r.t. | with respect to |

### Technical Abbreviations

| | |
|---|---|
| DEQ | Differential Equation |
| ECI | Earth centered inercial coordinate frame |
| ECEF | Earth centered Earth fixed coordinate frame (rotating with the Earth) |
| IDE | Integrated Development Environment |
| JAT | Java Astrodynamics Toolkit |
| JDK | Java Development Kit |
| JRE | Java Runtime Environment |
| JVM | Java Virtual Machine |
| MMH | Monomethylhydrazine |
| MMI | Man-machine Interface |
| OSK | *OpenSimKit* |
| SW | Software |
| XML | Extensible Markup Language |

# 2 Introduction

*OpenSimKit* is a free System Simulation software used for teaching purposes at the University of Stuttgart, Germany. From its historic roots its application to spacecraft engineering is by far the most mature domain.

The very first core simulator was implemented in FORTRAN 77 in the eighties [1], later was converted to an object oriented Kernel in C++ mid of the nineties [2] and was made available for student's use in 2004 by the original author.

Begin of 2008 the toolkit was converted to Java language by one of the former students, Alexander Brandt, to make it even more generic and simple to use for students getting in touch with a simulator for the first time. Applying Java all the nasty pointer and references handling topics belonged to the past, the build process was significantly simplified for a user and finally also the simulator now is independent from the underlying operating system preferred by the individual user.

This development lead to the current issue and the former C++ version meanwhile is deprecated and removed from the webpage. For historic milestones please also refer to the web page and section 20 . *OpenSimKit* is open source software and can be down-loaded for free from:

www.opensimkit.org

For topics of license, warranty and trademark of *OpenSimKit* please refer to section 21 .

As being based on Java *OpenSimKit* runs on most computer platforms and a large variety of operating systems. The development team always verifies proper functionality on MS Windows and on 32Bit and 64 bit Linux systems. The current versions *OpenSimKit* (OSK) consist of two separate programs intercommunicating via 2 sockets,

- the simulator osk-j-sim and
- the graphical man machine interface osk-j-mmi.

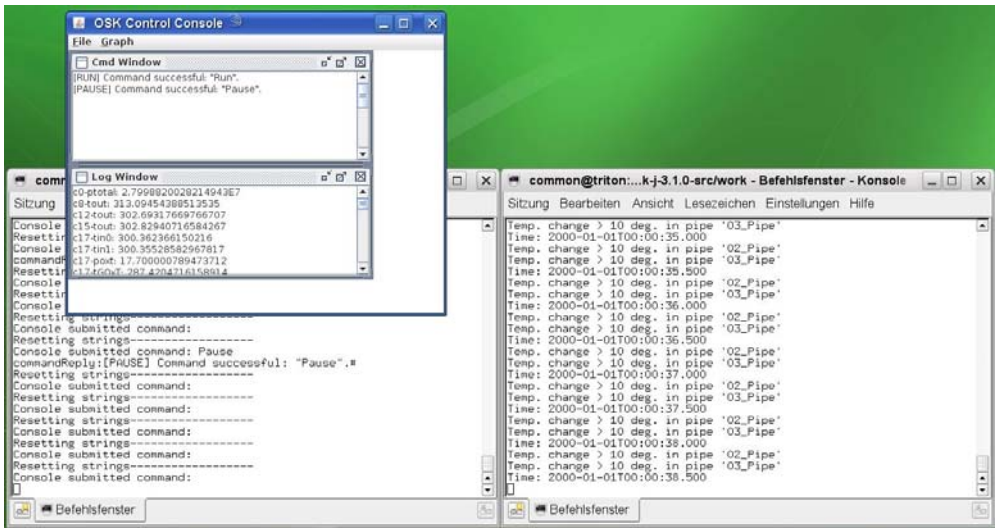



*Figure 1: MMI console window (left) and simulator console ( right) plus MMI main window.*

Both programs are started from ASCII consoles - independent of the operating system used on the host computer. The simulator process during run reports certain progress

information (e.g. actual time) to the ASCII console, but the simulator does not open up any further graphics window. The MMI opens the main graphics window from where the simulator can be commanded via mouse & menu functions. Also result plot & log sub-windows are provided here. In addition during run the MMI process logs certain progress info into its start-up ASCII terminal. See also the figure with MMI & console snapshots above.

Since Version 3.5.0 *OpenSimKit* in addition offers an interface to the astrodynamics visualization toolkit Celestia [6] for displaying simulated spacecraft in 3D in Orbit. Details on this functionality are given in Part III of this manual.

*OpenSimKit* performs simulation of technical/scientific systems which are modeled via mathematical differential equation systems. In principle it is suited for simulation of a large variety of systems, such as power plants, automobiles as well as for aerospace applications like rockets and satellites.

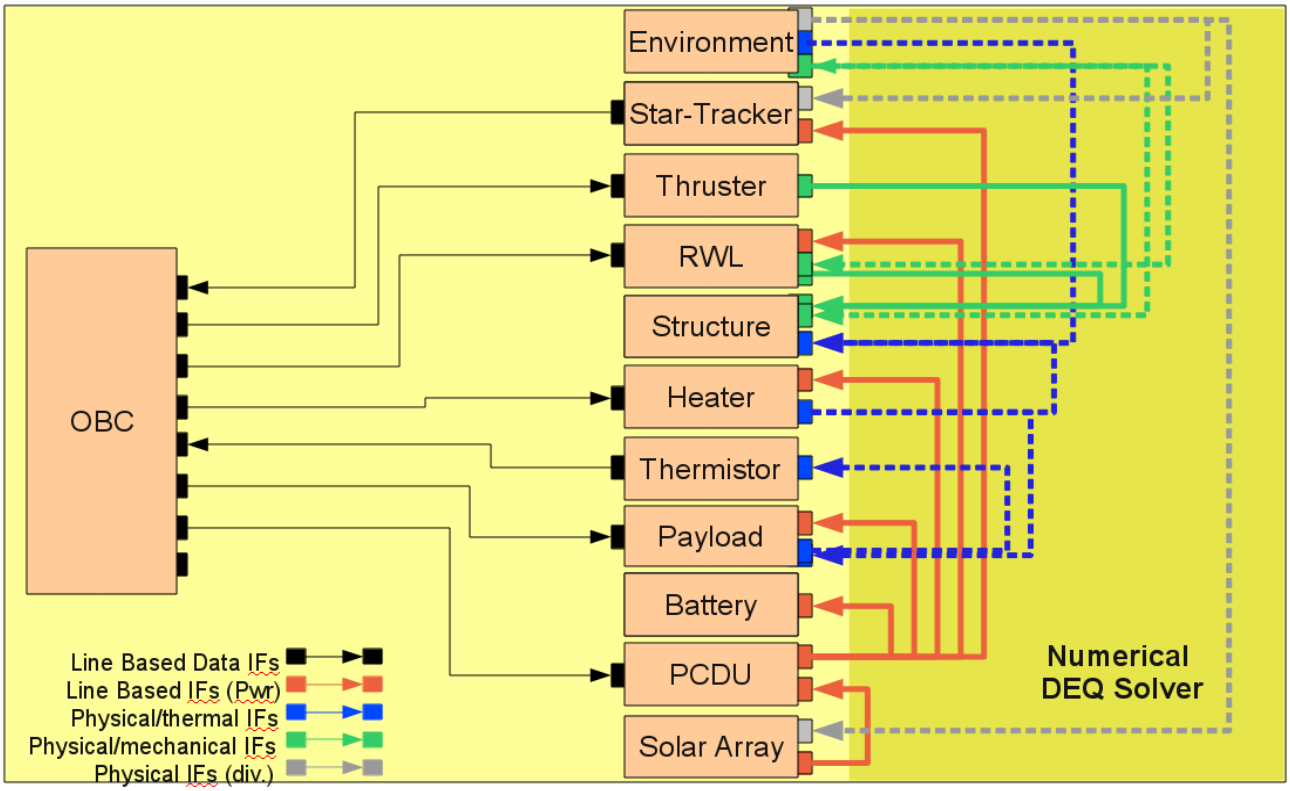



*Figure 2: Components and Interactions (Example: Satellite Modeling, from [3])*

*OpenSimKit* provides both solving of initial value problems (e.g. spacecraft orbit propagation) as well as boundary value problems and their combination (e.g. physics of spacecraft propulsion systems or space station life support systems). The important upgrade implemented in *OpenSimKit* since Version 3.7.0 is to determine between physical interaction of models based on real line or pipe interconnections and those which have no direct connection “port”. In the above system diagram there exist physical data lines between the onboard computer and certain satellite equipment. Similarly there exist real power lines between satellite equipment, the power control and distribution unit (PCDU) or between PCDU and Battery and Solar Array.

Such connections can be modeled via so-called port connections which are strongly typed. However gravity is computed by the space environment model but there is no “plug” connection imposing it onto the spacecraft structure. Furthermore such abstract parameters might be used by multiple client models – gravity might be required by several models.

To avoid defining fictive line connections between e.g. environment model and structure (as in Simulink®) *OpenSimKit* V3.7.0 and higher versions provide a so-called provider/ subscriber mechanism. Models computing abstract physical data provide these in according input file table entries and models using them place a subscriber entry. To a provided variable there might exist multiple subscribers. Models may use different local variable names. Variable typechecking is assured.

The system example provided together with the simulator infrastructure via a.m. Website represents a modeled re-ignitable medium energetic rocket upper stage as it is e.g. used for positioning satellites in highly inclined earth orbits. A flowchart of the system is given in Figure 6. Details on this system and its models are given in section 11 of this manual.

# 3 Hardware and Operating System Prerequisites

*OpenSimKit* was tested on Windows XP / Vista / 7 and diverse Linux platforms. It needs at least

- the JDK (Java Development Kit) 1.5 (preferably 1.6 - now called Java 6) and
- Apache Ant 1.7.0 (or higher).

*OpenSimKit* installs into a user defined directory and does not install any Windows DLLs , Linux .so libraries or similar. Therefore complex package based installation techniques are not required.

# 4 Software Dependencies

From release 2.4.6 *OpenSimKit* onwards, now each *OpenSimKit* release comprises in its installation .zip file all necessary dependencies which are required to run the distribution. Currently *OpenSimKit* has the following dependencies.:

- Woodstox 4.0.4:
  Woodstox is a Stax parser implementation which is needed, as the J2SE Runtime Environment (JRE) 1.5 does not include a Stax parser.
- SLF4J 1.5.6:
  The Simple Logging Facade for Java or (SLF4J) is an abstraction for various logging frameworks. *OpenSimKit* uses it together with Logback.
- Logback 0.9.15:

The back-end used for logging inside *OpenSimKit*. It can be extensively configured.

- Janino 2.5.15: Janino is an embedded Java compiler needed by logback.
- Extracts from JAT:
  The Java Astrodynamics Toolkit is used in a very bare version which supports only the gravity model. This is necessary as the complete JAT is very huge library which has itself lots of dependencies.
- Packet Library:
  Used for the communication between the MMI and the Simulator. However, the Packet Library is an *OpenSimKit* subproject and therefore delivered as source code and completely included into the build process.
- Junit 4.6:
  JUnit is used as a framework for the unit tests included in the *OpenSimKit* source distribution.
- JFreeChart 1.0.12:
  JFreeChart is necessary for plot windows.

These software libraries are directly included in the *OpenSimKit* downloads and **do not need to be installed separately by the user**. If the user already has other versions of these libs installed, *OpenSimKit* will use its own ones during the build process to avoid conflicts. The advantage is that *OpenSimKit* can be fully deinstalled simply by deleting its installation directory.

# Part I

# 5 Downloading & Installing *OpenSimKit*

When downloading *OpenSimKit* from the website, the user receives a .zip archive file which may be copied to the installation directory - e.g.

```
/home/OpenSimKit.
```

Unpacking the .zip for an *OpenSimKit* release osk-j-X.Y.Z-src.zip will lead to generation of a subdirectory ./osk-j-X.Y.Z-src in the selected installation directory. By this means multiple *OpenSimKit* releases easily can be installed aside of each other.

Unpacking the zip archive leads to the following substructure for binaries, libraries and input files within the ./osk-j-X.Y.Z-src directory:

```
osk-j-x.y.z-src\      Root directory of the OSK-J distribution.
|
|-- bin               Location of the jar files of both the MMI
|                     and the simulator.
|
|-- build             Location of the ant build script
|                     (build.xml).
|
|-- dependencies      Location of the dependencies needed to
|    |                compile the MMI and/or the simulator.
|    |-- janino-2_5_15
|    |                An embedded Java compiler needed by
|    |                logback.
|    |-- jfreechart-1_0_12
|    |                  Location of the jfreechart and jcommon
|    |                jar file necessary for plotting graphs.
|    |-- junit-4_6    Location of the JUnit jar file in version 4.6.
|    |-- logback-_0_9_15
|    |                 Logging back-end used by OSK-J.
|    |-- osk-j-jat-minimum_1_0_0 Minimum version of JAT for OSK-J.
|    |                Contains only classes necessary for gravity
|    |                model.
|    |-- slf4j-1_5_6  The logging interface used in OSK-J.
|    |
|    |-- woodstox-4_0_4
|                      Location of the woodstox jar file in
|                     version 4.0.4.
|
|-- lib               Location of libraries needed by the
|                     simulator and/or MMI and/or models.
|-- mmi
|    |-- src          MMI source code directory.
|    |
|    |-- tests        MMI unit test code directory.
|
|-- models            Location of the model libraries. Starting
|                     with release 2.5 the simulator does not
|                     include any simulation model classes.
|                     Instead it searches for jar files containing
|                     model classes in this directory.
|-- netbeans          Netbeans free-form projects. One for each
|                     module (mmi, pkt, rpr, sim).
|-- pkt
```

```
|    |-- src          Packet Library source code directory.
|    |
|    |-- tests        Packet Library unit test code directory.
|
|-- rpr
|    |-- src          Rocket propulsion system source code
|    |                directory.
|    |-- tests        Rocket propulsion system unit test code
|                     directory.
|-- sim
|    |
|    |-- src          Simulator source code directory.
|    |
|    |-- tests        Simulator unit test code directory.
|
|-- temp              Location of the temporarily created files
|                     during compilation.
|-- work              Location of the simulator input files.
```

For the beginner user, the ./work and the ./build subdirectories are of primary interest to build and start up the software. Please see the following sections explaining software build and startup.

Since software version V3.3 example input files are provided to the user directly in the ~/work subdirectory. Furthermore from V3.3 due to distribution file size limits on the hosting server, the software and the documentation are split into two downloads.

From V3.5 onwards a third, optional download file is provided containing all elements for the coupling of *OpenSimKit* to the astrodynamics visualization software Celestia. For more details please refer to Part III of this manual.

# 6 Building *OpenSimKit* Simulator and MMI Program

To compile *OpenSimKit* the program "Apache Ant" is needed. If it is not already present on your system you can download it from http://ant.apache.org/ . Please note that the *OpenSimKit* build system is currently only tested with Ant 1.7.0 and higher versions. To compile *OpenSimKit* open a terminal console on your computer and navigate into the *OpenSimKit* distribution's "`./build`" subdirectory.

Principally the build system has multiple targets, i.e. simulator, MMI etc. can be built individually, but this feature is postponed to a later chapter here. For the beginner user the simple command

```
>ant all
```

builds all binaries in the archive including the according javadoc documentation. This is what most newcomers will do first. The build process will report building of the diverse targets onto the text console. A message similar to

```
BUILD SUCCESSFUL
Total time: 4 seconds
```

will document the successful compilation of all targets and the proper arrangement of all configuration files in the work directory.

# 7 Quick Start Information

To run a simulation the simulator must be started first. It will start up and wait listening to communication sockets 1500 and 1510 for a Man Machine Interface to connect to it. So after simulator startup the MMI has to be started - which automatically searches for a booted simulator and connects to it - and then from the MMI the simulation can be started, stopped, controlled, terminated etc. These steps are explained now in the subsequent subchapters.

## 7.1 Starting Simulation and MMI

To start up the simulator go into the ./work directory of the *OpenSimKit* distribution installation and type:
Command line Windows:

```
osk-j-sim.bat [inputfile.xml] [outputfile.txt]
```

Command line Linux:

```
./osk-j-sim.sh [inputfile.xml] [outputfile.txt]
```

- The inputfile `Rocket-Stage-Simulation.xml` is distributed together with the *OpenSimKit* standard installation comprising the rocket simulation library as

example.

- The outputfile is the text file where the simulation will log its output data. The selection of what results at which timesteps are to be logged can be specified in the inputfile - details see later).

After the simulation is up and running and is waiting for an MMI to connect, the MMI can be started up by:

Commandline Windows:

```
osk-j-mmi.bat 127.0.0.1
```

Commandline Linux:

```
./osk-j-mmi.sh 127.0.0.1
```

## 7.2 *OpenSimKit* MMI

After the simulation has been started up according to chapter 7.1 , e.g. by:

```
./osk-j-sim.sh Rocket-Stage-Simulation.xml myOutFile.txt
```

and after the simulation MMI has been started according to chapter 7.1 , the simulation computation and results visualization can be controlled via the MMI. Figure 4 depicts a screenshot of the MMI main window:

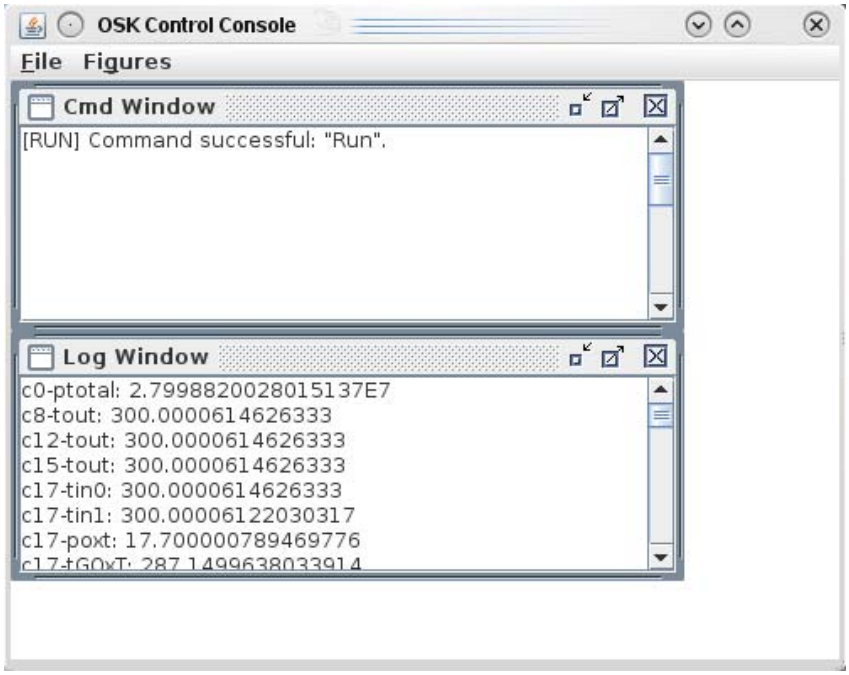



*Figure 3: OpenSimKit MMI main window*

The MMI main window comprises two subwindows. The upper subwindow displays the

submitted commands and their state and the lower subwindow displays the logged parameters sent to the MMI by the simulator via a packet transfer mechanism during running simulation. The MMI menu bar comprises a File and a Figures menu:

- File menu: Four buttons representing the most important *OpenSimKit* simulator control commands and a button to shutdown the MMI are included. Shutdown of the MMI automatically implies shutdown of the simulation too.
  - Click on the menu item "File" and then on the menu item "Run" to start the simulation computation.
  - Now you can enter commands in the "Cmd Window". Just enter your commands in the command window (Named "Cmd Window") in the last empty line and hit the enter key to send the command. For the results of your commands take also a look at the console output in addition to the "Log Window".
  - Click on "File|Stop" to pause the simulator.
  - Click on "File|Resume" to resume the paused simulator.
  - Click on "File|Shutdown" to terminate the MMI and the simulator.
  - For a list of available commands take a look at chapter 9 Command Reference.

The simulation generates an outputfile - in this example named `myOutFile.txt` - into which it cyclically log the output results of the computation, according those the variables selection and log frequency settings in the inputfile. These logged parameters can be visualized in diverse ways using the next MMI menu:

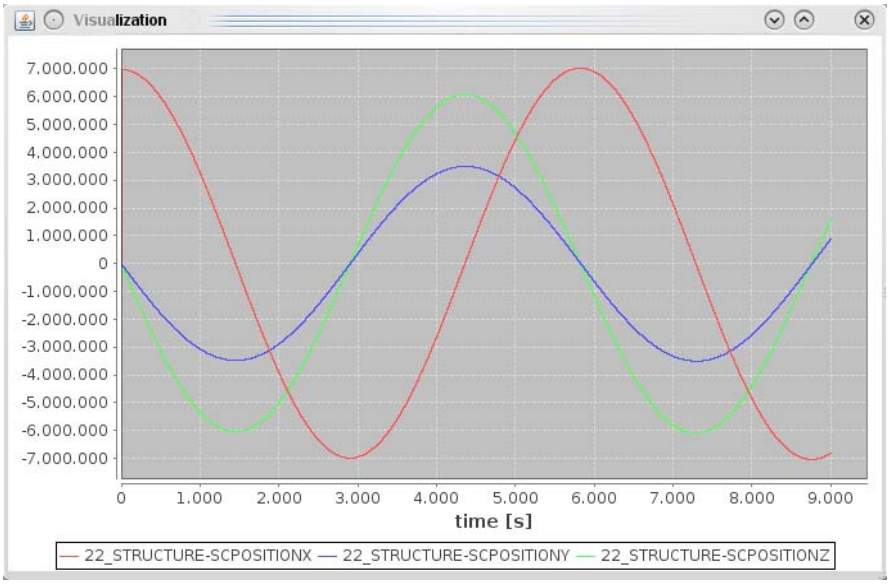



*Figure 4: MMI plot window with plotted simulation results*

- Figures menu:
  - The first menu entry can be used to locate a simulation output file of a completed or still running simulation.
  - The second entry opens a dialog box for selection of parameters from the outputfile which are to be plotted over time together in one plot window. An example of such a plot is given by Figure 4. Multiple such plot windows with different parameter sets may be opened in parallel.
  - The third menu entry opens a ground track plot window depicting the

spacecraft's position evolution over an earth map - see Figure 5.

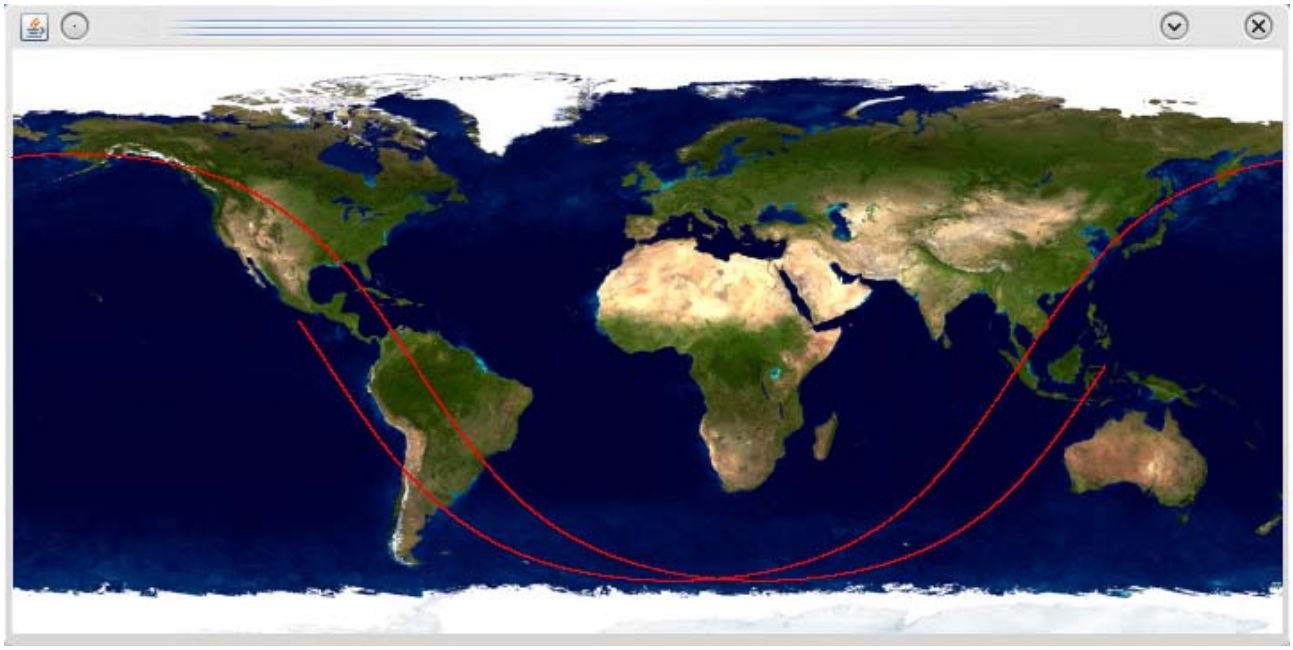


*Figure 5: MMI spacecraft ground track plot over Earth map.*

Since V3.6.0 of *OpenSimKit* the plot windows of the MMI are automatically updated with new parameter results written to the OSK result output file. So parameter values evolving over time can be monitored.

The plot granularity depends on the output stepsize selected for the simulator result logging to the output file. The latter is an adjustable parameter in the simulation run input file.

# 8 Structure of an *OpenSimKit* XML Input File

The simulation input files consist of the following five sections:

- system
- models
- connections
- meshes
- logOutput.

Which shall be described briefly on the following pages. To avoid this section to become too theoretic, please in parallel have a look at one of the provided .xml input files included in the "`./work`" subdirectory.

## 8.1 OpenSimKitConfigFile

The root element of an *OpenSimKit* XML configuration file.

**Containing elements:**

system, models, Fehler: Referenz nicht gefunden, meshes, logOutput

**Attributes:**

| Attribute | Type | Required | Description |
|---|---|---|---|
| xmlns | string | TBD | TBD |
| xmlns:xsi | string | TBD | TBD |
| xsi:schemaLocation | string | TBD | TBD |

## 8.2 system

This element contains all information about the simulator setup.

**Containing elements:**

The section system of the input files consists of three subsections:

- Sysdesc
- RunSettings
- SimulationCtrl

**Attributes:**

TBD

## 8.3 models

The “models” section is used to specify the Models which are used in this particular simulation. For each Model a separate subsection is created. This subsection contains information for the particular Model.

**Containing elements:**

`model (for model definitions)`

**Attributes:**

none

### 8.3.1 model (for model definitions)

This element contains all information about the initialisation of one model instance.

**Containing elements:**

`variable`

**Attributes:**

| Attribute | Type | Required | Description |
|---|---|---|---|
| class | string | yes | Fully qualified name of the class from which this model will be created. |
| name | string | yes | Unique name of the model. |

### 8.3.2 variable

This element contains all information to initialise one model variable.

**Containing elements:**

none

**Attributes:**

| Attribute | Type | Required | Description |
|---|---|---|---|
| description | string | no | Contains a description of this variable. |
| length | positive integer | no | Specifies the size of the array. Necessary if this variable is an array. The array counting starts with zero. |
| name | string | yes | Unique name of the model. |
| same | boolean | no | Specifies that all array elements are the same. This avoids unnecessarily repeating the same value for each element of an array. |

| | | | |
|---|---|---|---|
| state | string | no | Used to describe if this variable is an input or output variable. |
| unit | string | yes | Contains the unit of the variable. |

### 8.3.3 Model Section Example

A model definition is shown here:

```
<model class="fully.qualified.Classname" name="name of this model">
  <variable name="variable1">value</variable>
  <variable name="arrayVariable1" length="3">value1 value2 value3</variable>
  <variable name="arrayVariable2" length="4" same="true">value</variable>
</model>
```

An example:

```
<model class="org.opensimkit.model.rocketpropulsion.PRegT1" name="08_PReg">
  <variable name="description"/>
  <variable name="length" unit="m">.1</variable>
  <variable name="mass" unit="kg/m">2.6</variable>
  <variable name="innerDiameter" unit="m">.014</variable>
  <variable name="specificHeatCapacity">900.0</variable>
  <variable name="pcoeff" length="4">24.10245 .4462006 -1.84912E-3 2.580329E-6
  </variable>
  <variable name="temperature">300.0</variable>
</model>
```

Each model definition has the tag “model”. It has the two mandatory attributes “class” and “name”. The attribute class needs the fully qualified name of the class from which this model will be created. In this case it is “org.opensimkit.models.rocketpropulsion.PRegT1”. It means the class “PregT1” from the package “org.opensimkit.models.rocketpropulsion”. The fully qualified class name is necessary, because it is used by the OpenSimKit kernel to instantiate an object of the class. The attribute “name” denotes the name of the Model. Each Model must have a unique name. In the example above the Model's name is “08_PReg”. The manipulatable variables of the Model are written between the tags of “model”. The variables with the names length, mass, innerDiameter, specificHeatCapacity, and temperature are scalar variables. They are written in the same way as in the source code of the PregT1 class. The value of the variable must be compatible with the variable type inside the Java class. It is also possible to write additional attributes to this XML tag, but they are currently ignored by the OpenSimKit XML parser. The variable pcoeff is an one-dimensional array. It has the mandatory attribute “length” which denotes the length of the array. The text of this tag includes the values delimited by at least one space.

A second example:

```
<model class="org.opensimkit.models.rocketpropulsion.PipeT1 name="02_Pipe">
  <variable name="description"/>
  <variable name="length" unit="m">1.5</variable>
  <variable name="specificMass" unit="kg/m">.6</variable>
  <variable name="innerDiameter" unit="m">.0085</variable>
  <variable name="specificHeatCapacity">500.0</variable>
  <variable name="surfaceRoughness" unit="m">1.E-6</variable>
  <variable name="temperatures" length="10" same="true">300.0</variable>
</model>
```

The important variable in this example is the temperatures variable. As it is visible by the attribute “length” the temperatures variable is a one-dimensional array. The interesting thing to know is the “same” attribute. It denotes that all array values are the same and as such there needs only one value to specified. The OpenSimKit XML parser automatically initializes the array with the correct values and frees the user from unnecessary writing ten times the same value into the configuration file.

## 8.4 connections

This element contains all information about the connections between the models.

**Containing elements:**

`connection`

**Attributes:**

none

### 8.4.1 connection

This element contains all information about the initialisation of one model instance.

**Containing elements:**

`Fehler: Referenz nicht gefunden, Fehler: Referenz nicht gefunden, Fehler: Referenz nicht gefunden`

**Attributes:**

| Attribute | Type | Required | Description |
|---|---|---|---|
| class | string | yes | Fully qualified name of the class from which this connection will be created. |
| name | string | yes | Unique name of the connection. |

### 8.4.2 Connection Section Example

A connection definition is shown below:

```
A connection definition is shown here:
    <!-- Connection of Pipe to Filter. -->
    <connection name="05_PureGasDat" class="org.opensimkit.ports.PureGasPort">
      <from model="05_Pipe"     port="outputPort"/>
      <to   model="06_Filter"   port="inputPort"/>
    </connection>
```

## 8.5 providerSubscriberTable

This element contains all information about variable interchange between models which is not subject to transfer via modeled real electrical connections, fluid pipes or similar.

between the models.

**Containing elements:**

`entry`

**Attributes:**

none

### 8.5.1 entry

This element contains information about one provider/subscriber table entry.

**Containing elements:**

`Fehler: Referenz nicht gefunden, provider, subscriber`

**Attributes:**

| Attribute | Type | Required | Description |
|---|---|---|---|
| provider | string | yes | Fully qualified name of the class from which this connection will be created. |
| name | string | yes | Unique name of the connection. |

### 8.5.2 ProviderSubscriber Table Section Example

A providerSubscriber definition is shown below:

```
<entry name="gravityAcceleration">
  <!-- Type = double[4] -->
  <provider   model="23_Environment"  variable="gravAcceleration"/>
  <subscriber model="22_Structure"  variable="gravityAccel"/>
</entry>
```

Note: Vectors like given gravity example are defined as double[4]. The first vector element contains the magnitude, the other 3 the normalized direction components in the corresponding coordinate system.

## 8.6 mesh (for mesh definitions)

This element contains all information about the meshing of the models.

**Containing elements:**

`mesh`

**Attributes:**

| Attribute | Type | Required | Description |
|---|---|---|---|
| name | string | yes | Unique name of the mesh. |
| level | string | yes | The level of the mesh. “top” for the top-level mesh and “sub” for the sub-level meshes. |

### 8.6.1 meshes

This element contains all information about the meshing of the models for the boundary condition solver.

**Containing elements:**

`mesh (for mesh definitions)`

**Attributes:**

none

### 8.6.2 Mesh Section Example

A nested mesh definition is shown below:

```
<meshes>
    <mesh name="mesh_0" level="top">
        <model>21_EngineController</model>
        <mesh>mesh_1</mesh>
        <model>22_Structure</model>
        <model>23_Environment</model>
    </mesh>
    <mesh name="mesh_1" level="sub">
        <mesh>mesh_2</mesh>
        <model>05_Pipe</model>
        <model>06_Filter</model>
        <model>07_Pipe</model>
        <model>08_PReg</model>
        <model>09_Pipe</model>
        <model>10_Split</model>
        <model>11_Pipe</model>
        <model>12_PReg</model>
        <model>13_Pipe</model>
        <model>14_Pipe</model>
        <model>15_PReg</model>
        <model>16_Pipe</model>
        <model>17_Tank</model>
        <model>18_FluidFlowValve</model>
        <model>19_FluidFlowValve</model>
        <model>20_Engine</model>
    </mesh>
    <mesh name="mesh_2" level="sub">
        <model>00_HPBottle</model>
        <model>01_HPBottle</model>
        <model>02_Pipe</model>
        <model>03_Pipe</model>
        <model>04_Junction</model>
    </mesh>
</meshes>
```

## 8.7 logOutput

This element contains all information about the logging of model variables to the output file.

**Containing elements:**

Fehler: Referenz nicht gefunden

**Attributes:**

| Attribute | Type | Required | Description |
|---|---|---|---|
| delimiter | string | yes | The delimiter between two entries. The string “\t” stands for the tabulator. |
| end | string | yes | The time for which the logging will end. |
| factor | positive integer | yes | Every “factor” iterations an entry is written to the output file. |
| start | string | yes | The time for which the logging will start. |

### 8.7.1 entry

This element contains information about one logfile entry.

**Containing elements:**

none

**Attributes:**

| Attribute | Type | Required | Description |
|---|---|---|---|
| format | string | no | How the output value should be formatted. All Java formatting options are supported (see http://java.sun.com/j2se/1.5.0/docs/api/java/util/Formatter.html). |
| header | string | no | The header of the output values. If this attribute is missing then the name of the variable is used for the header. |
| model | string | yes | Name of the model. |
| variable | string | yes | Name of the model's variable, which should be written to the output file. |

### 8.7.2 LogOutput Section Example

A log output definition is given below:

```
  <logOutput start="0.0" end="250.0" factor="1" delimiter="\t">
    <entry model="timeHandler" variable="simulatedMissionTime" header="Mission
Time" format="%1$tFT%1$tH:%1$tM:%1$tS.%1$tL"/>
    <!--
    <entry model="00_HPBottle" variable="ptotal" format="%6.6f"/>
    <entry model="01_HPBottle" variable="ptotal" format="%6.6f"/>
```

```
    <entry model="00_HPBottle" variable="mftotal" format="%6.6f"/>
    <entry model="01_HPBottle" variable="mftotal" format="%6.6f"/>
    <entry model="20_Engine" variable="thrust" format="%6.6f"/>
    <entry model="20_Engine" variable="alt" format="%6.6f"/>
    <entry model="22_Structure" variable="scVelocityX" format="%6.6f"/>
    <entry model="22_Structure" variable="scVelocityY" format="%6.6f"/>
    <entry model="22_Structure" variable="scVelocityZ" format="%6.6f"/>
    <entry model="22_Structure" variable="scPositionX" format="%6.6f"/>
    <entry model="22_Structure" variable="scPositionY" format="%6.6f"/>
    <entry model="22_Structure" variable="scPositionZ" format="%6.6f"/>
    <entry model="22_Structure" variable="scPosLat" format="%6.6f"/>
    <entry model="22_Structure" variable="scPosLon" format="%6.6f"/>
    <entry model="22_Structure" variable="scPosAlt" format="%6.6f"/>
</logOutput>
```

# 9 *OpenSimKit* Command Reference

## 9.1 How to enter Commands

Enter your commands in the command window (named "Cmd Window") in the last empty line and hit the enter key to send the command. The commands themselves are not case sensitive, however, the arguments can be case sensitive.

Note: A subset of the available commands are intuitively accessable via the MMI's pulldown menus.

## 9.2 Short Command Overview

These are all currently implemented simulator commands:

```
call <instance> <method>
disconnect
get <instance> <field>
geta <instance> <field> <position>
help
resume
run
set <instance> <field> <value>
seta <instance> <field> <position> <value>
shutdown
stop
```

## 9.3 Extended Command Description

### 9.3.1 Call

```
call <instance> <method>
```

Calls a callable method inside the simulator. Currently only parameterless methods are supported.

<instance> is the name of the component instance you want to address (in case a class is instantiated more than once). It is case sensitive. Use the help command to get a list of all available manipulatable variables.

<method> is the name of the method of the instance of which you would like to execute. It is case sensitive.

Examples:

```
call c0 init
```

### 9.3.2 Disconnect

```
disconnect
```

Disconnects the MMI from the simulator.

### 9.3.3 Get

```
get <instance> <field>
```

Returns the value of a manipulatable variable.

<instance> is the name of the component instance you want to adress (in case a class is instantiated more than once). It is case sensitive. With the supplied sample files there exist 20 components named “00_HPBottle” to “19_FluidFlowValve”. However, not all have manipulatable variables. Use the help command to get a list of all available manipulatable variables.

<field> is the name of the member variable of the instance of which you would like to get its value. It is case sensitive. The following fields are in the instance "00_HPBottle":

```
mass                    double variable
volume                  double variable
specificHeatCapacity    double variable
ptotal                  double variable
ttotal                  double variable
fluid                   String variable
```

Examples:

```
get 00_HPBottle mass
```

### 9.3.4 GetA

```
geta <instance> <field> <position>
```

Returns the value of an element of an one-dimensional array.

<instance> is the name of the component instance you want to address (in case a class is instantiated more than once). It is case sensitive. With the supplied sample files there exist 20 components named “00_HPBottle” to “19_FluidFlowValve”. However, not all have manipulatable variables. Use the help command to get a list of all available manipulatable variables.

<field> is the name of the member variable of the instance of which you would like to get its value. It is case sensitive. The following fields are in the instance "02_Pipe":

```
innerDiameter           double variable
length                  double variable
```

```
specificMass            double variable
specificHeatCapacity    double variable
surfaceRoughness        double variable
temperatures            Array of double variable (length 10)
qHFlow                  Array of double variable (length 10)
massPElem               double variable
pin                     double variable
tin                     double variable
mfin                    double variable
pout                    double variable
tout                    double variable
```

<position> is the position in the one-dimensional array.

Examples:

```
geta 02_Pipe temperatures 0
geta 02_Pipe temperatures 9
geta 02_Pipe temperatures 10  (will print an error message, as
the first array element is called 0)
geta 02_Pipe qHFlow 9
```

### 9.3.5 Help

```
help
help <command name>
help <component name>
help <simulator information>
```

The Help Command prints information about the simulator.

help: Invoked without any parameter it prints a list of all available commands.

help <command name>: Prints a detailed help about the command.

help <component name>: Prints detailed information (Manipulatables, Readables, Callables) about the Component.

help <simulator information>: Prints specific global simulator information.
Possible values are:

Components: A list of the names of all Components inside the simulator as well as their type.

Manipulatables: A list of all Manipulatables inside the simulator ordered by classes as well as their type.

Readables: A list of all Readables inside the simulator ordered by classes as well as their type.

Callables: A list of all Callables inside the simulator ordered by classes.
Version: Prints the version of the simulator.

Examples:

```
help
help get
help 00_HPBottle
help components
help manipulatables
help readables
help callables
help version
```

### 9.3.6 Resume

```
resume
```

Resumes a stopped simulation. This command is also available as a menu item (File| Resume).

### 9.3.7 Run

```
run
```

Starts a simulation. This command is also available as a menu item (File|Run).

### 9.3.8 Set

```
set <instance> <field> <value>
```

Sets the value of a manipulatable variable.

<instance> is the name of the component instance you want to adress (in case a class is instantiated more than once). It is case sensitive. With the supplied sample files there exist 20 components named “00_HPBottle” to “19_FluidFlowValve”. However, not all have manipulatable variables. Use the help command to get a list of all available manipulatable variables.

<field> is the name of the member variable of the instance of which you would like to get its value. It is case sensitive. The following fields are in the instance “00_HPBottle”:

```
mass                  double variable
volume                double variable
specificHeatCapacity  double variable
ptotal                double variable
ttotal                double variable
fluid                 String variable
```

<value> is the value the field should be set to.

Examples:

```
set 00_HPBottle mass 1000
set 00_HPBottle fluid test
```

### 9.3.9 SetA

```
seta <instance> <field> <position> <value>
```

Sets the value of an element of an one-dimensional array.

<instance> is the name of the component instance you want to adress (in case a class is instantiated more than once). It is case sensitive. With the supplied sample files there exist 20 components named “00_HPBottle” to “19_FluidFlowValve”. However, not all have manipulatable variables. Use the help command to get a list of all available manipulatable variables.

<field> is the name of the member variable of the instance of which you would like to get its value. It is case sensitive. The following fields are in the instance “02_Pipe”:

```
innerDiameter           double variable
length                  double variable
specificMass            double variable
specificHeatCapacity    double variable
surfaceRoughness        double variable
temperatures            Array of double variable (length 10)
qHFlow                  Array of double variable (length 10)
massPElem               double variable
pin                     double variable
tin                     double variable
mfin                    double variable
pout                    double variable
tout                    double variable
```

<position> is the position in the one-dimensional array.

<value> the value of the element.

Examples:

```
seta 02_Pipe temperatures 0 300
seta 02_Pipe temperatures 9 -100
seta 02_Pipe temperatures 10 30000  (will print an error
message, as the array starts at position 0)
seta 02_Pipe qHFlow 9 20
```

### 9.3.10 Shutdown

```
shutdown
```

Stops the simulation and closes the simulator as well as the MMI. This command is also available as a menu item (File|Shutdown).

### 9.3.11 Stop

```
stop
```

Stops a running simulation. This command is also available as a menu item (File|Stop).

# 10 De-installing *OpenSimKit*

To de-install *OpenSimKit* simply remove the entire *OpenSimKit* release directory from your computer, i.e. the ./osk-j-X.Y.Z-src directory. *OpenSimKit* does not create any files outside this directory structure.

# Part II

# 11 The Rocket Stage Model Library

This section provides an introduction to the physics implemented in the equipment models of the rocket stage simulation library which is included in the *OpenSimKit* standard distribution. This library models a rocket upper stage with a reingnitable engine powered by N2O4 and MMH. The data also fit with sufficient precision for N2O4 / Aerozine 50 oxidizer / fuel combinations as it was used by the former ELDO Astris stage. Part of the models from this library are descendants from [1] ad were verified with data from the ELDO Astris programme. The layout of the propulsion system is shown in Figure 6.

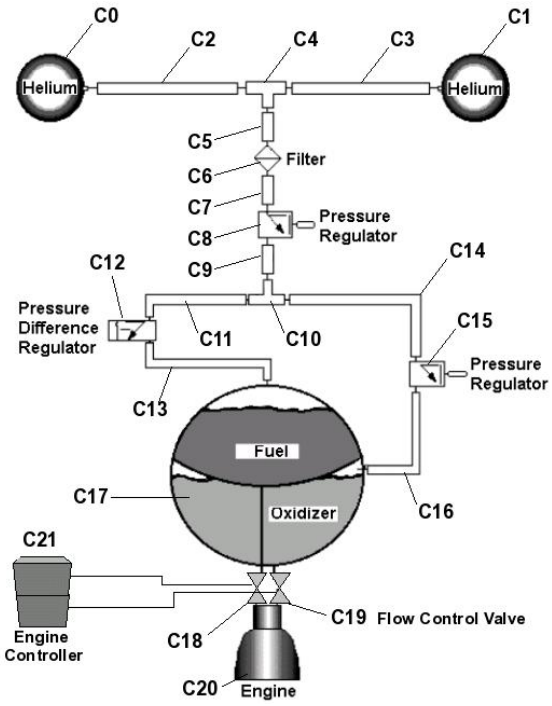



*Figure 6: Propulsion System Layout.*

Besides the component models of the pure propulsion system, the rocket stage model in addition comprises a structure model and a space environment model to be able to simulate rocket stage flight dynamics in space.

## 11.1 The Helium Gas Model

Helium is used frequently as inert gas to pressurize propulsion system tanks which are filled with hydrazine derivates and according oxidizers, since these substances are highly reactive. However when using Helium as pressure gas its non-idealness especially concerning its expansion behaviour has to be considered to compute quantitatively proper results. This can be achieved by reflecting in detail the variance of Helium's Joule-Kelvin coefficient as function of both pressure and temperature.

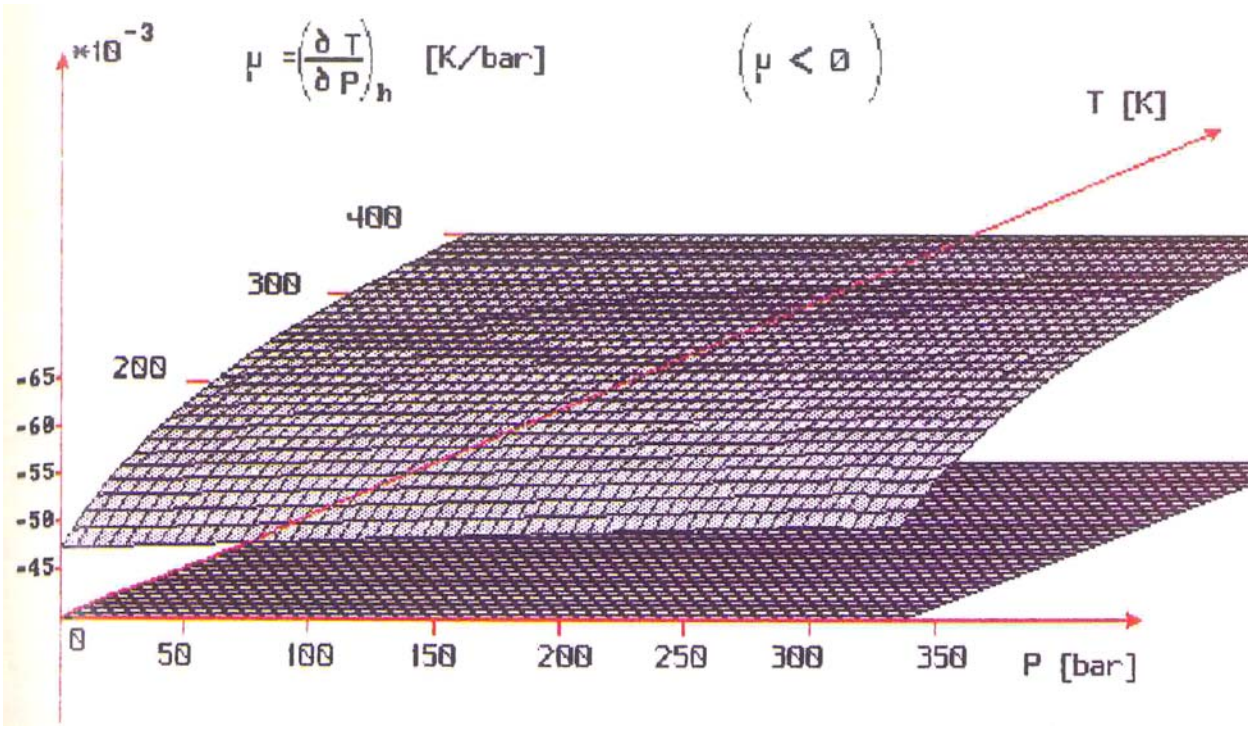



*Figure 7: Joule-Kelvin coefficient of Helium as function of pressure and temperature*

The Joule-Kelvin coefficient of Helium is required to compute its specific enthalpy. In the Helium gas model the JKC is modeled by a set of polynomes at 1 bar, 20 bar, 40 bar etc. up to 320 bar which model the JKC as function of temperature. The dependency of the JKC at a certain temperature over pressure can be modeled as linear interpolation.

The specific enthalpy of Helium at a reference pressure of 5 bar can be modeled as polynome:

$$h = -19846.5 + 5732.967\,T - 2.42982\,T^2 + 3.332099^{-3}\,T^3 \qquad (11.1)$$

If the Helium state is characterized by another pressure, instead of the actual temperature, a reference temperature has to be inserted in the above equation. This reference temperature is computed by the formula below and represents a state as if the gas were expanded / compressed to the target pressure - considering its non-idealness through the JKC:

$$T_{ref} = T - \mu\,(p - p_{ref}) \qquad (11.2)$$

This reference temperature then can be entered in formula 11.1 to compute the specific enthalpy of the Helium gas.

Another topic is the computation of the non ideal behavior of Helium at high pressure levels. The deviation from the ideal gas law can be reflected by a compressibility factor $Z$:

$$p = Z\,\rho\,R\,T \qquad (11.3)$$

Where

$$Z = 1.0 + F * p \tag{11.4}$$

with pressure $p$ is to be reflected in [bar] and

$$F = 1.913688^{-3} - 8.520942^{-6} T + 1.358845^{-8} T^2 - 4.595341^{-12} T^3 \tag{11.5}$$

## 11.2 The High Pressure Bottle Model

The software class HPBottleT1 represents the high pressure bottles holding the pressurant gas Helium for pressurizing the rocket fuel- and oxidizer tank. The temperature over time is computed according to the first law of thermodynamics:

$$dT = \frac{\dot{Q}\, dt}{m\, c_v} + \frac{\dot{m}\, T\, dt}{m} - \frac{h\, \dot{m}\, dt}{m\, c_v} \tag{11.6}$$

The cinetic energy of pressurant leaving the bottle can be neglected here [01]. The conductive heat transfer results from the temperature difference btw. fluid and bottle wall and is driven by the heat transfer coefficient:

$$\dot{Q} = \alpha\, A\, (T_w - T) \tag{11.7}$$

For computation of the pressure gase's specific enthalpy please refer to section 11.1 . Currently for integration of the DEQ 11.6 a simple explicit Euler method is used - which is acceptable since all gradients are negative and thus the method is always converging and stable.

$$T_{new} = T_{old} + dT \tag{11.8}$$

The gas pressure after finishing the timestep integration - i.e. after simulating a $dm$ of mass has vented from the bottle to its outlet - is computed under consideration of the non-ideal behaviour of the gas at high pressure levels. This is essential for Helium as pressurant. The deviation from the ideal gas law is reflected by the compressibility factor $Z$. The equation reads as follows:

$$p = Z\, \rho\, R\, T \tag{11.9}$$

The density ρ results from division of the remaining mass by the bottle volume. But the factor $Z$ itself is a function of the pressure. The dependency of $Z$ as function of p is computed in the pressurant Helium gas model through a polynomial approximation which is checked against experimental data [02]. See equations 11.4and 11.5. Applying this

method the remaining pressure in the bottle is computed iteratively. A start pressure from ideal gas expansion is presumed, the compressibility factor is computed and the resulting non ideal pressure is compared against the assumption. The pressure value is iteratively corrected until the precision is better than $5*10^{-4}$ .

The adequate equation to model thermal heat transfer between pressure gas and bottle wall is the one for free convection in sperical containments:

$$\alpha = Nu \frac{\lambda}{d} \qquad (11.10)$$

With:

$$Nu_d = 0.098 (Gr_d Pr)^{0.345} \qquad (11.11)$$

## 11.3 The Pipe Model

The PipeT1 model reflects a gas flow through a pipe, considering friction effects and heat transfer from pipe wall to fluid. Considering heat transfer to environment the pipe is modeled to be ideal adiabatic.

### 11.3.1 Modeling the Pressure Drop

To compute the pressure drop first the flow velocity inside the pipe has to be identified. Therefore first the so-called critical velocity of the flow is computed which depends on the in-flow gas temperature and is received via the input port of the pipe model. The flow velocity is computed in an iterative way:

- First a velocity is assumed, therefrom - using the critical velocity - the Laval-number is derived and then using the formulas for adiabatic gas expansion the pressure and temperature of the gas are derived.
- Then using the Helium gas model the gas density is computed, considering again non-ideal gas. And therewith the mass flow through the pipe can be identified.
- Then the presumed input flow velocity is iteratively adapted until the derived mass flow and the input mass flow at the pipe's input port differ by less than $10^{-3}$ kg/sec.

Using the flow velocity derived by this iteration, the Reynolds number is computed and then the friction factor $\lambda$ is to be computed.

$$\lambda = f(\log Re, \frac{ks}{d}) \qquad (11.12)$$

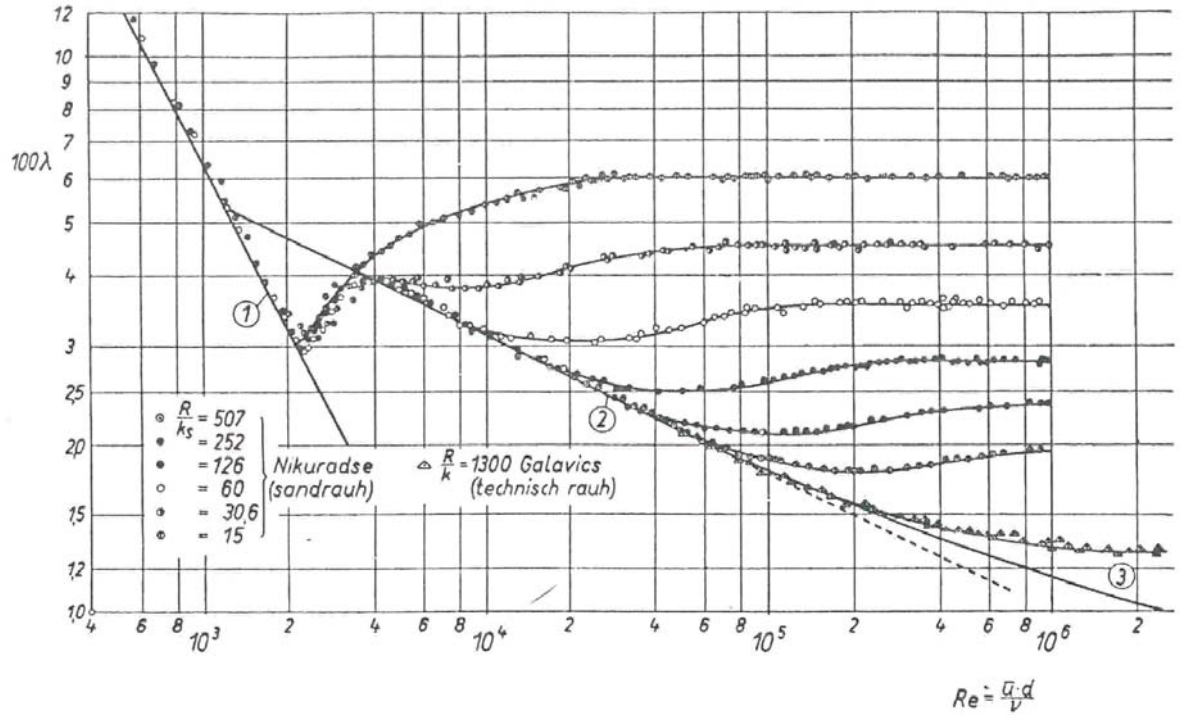



*Figure 8: Moody Diagram: Pipe Lambda coefficient as function of Reynolds Number*

Manually this could be done by interpolating in the Moody-diagram, for computation purposes an iterative solving of the Colebrook equation is performed.

$$1/\sqrt{\lambda} = -2\log_{10}((e/D)/3.71 + 2.51/(\mathrm{Re}\sqrt{\lambda}))$$

*or*

$$\lambda = 0.25/\left(\log_{10}\left(\frac{(e/D)}{3.71} + \frac{2.51}{(\mathrm{Re}\sqrt{\lambda})}\right)\right)^2 \qquad (11.13)$$

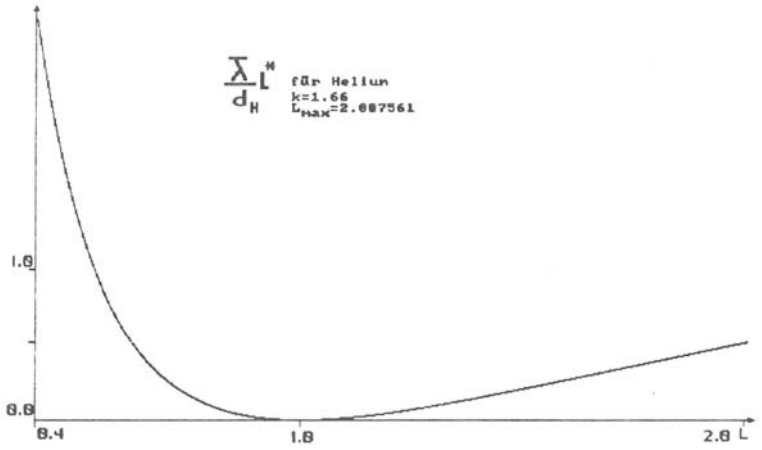



*Figure 9: Flow friction coefficient for adiabatic Helium flow in pipe.*

This approach is based on assuming pipes with TEFLON inliner and covers the transition range between laminar and entirely turbulent flow. Purely laminar flows thus are not covered by this approach, but since pipe inflow fittings, elbow connectors etc. will lead to at least partly turbulent flow conditions this implementation approach is valid.

With the available fluid density and the friction coefficient now the outlet pressure of the pipe can be computed via:

$$p_{out} = p_{in} - \frac{\rho}{2} v^2 \lambda \frac{l}{d} \qquad (11.14)$$

For reflecting the throttling effects and heat transfer from pipe wall to fluid the pipe is split into 10 subsections and for each element the subsequent temperature changes are computed individually. First the Nußelt number has to be computed, which is done applying:

$$Nu = \frac{\frac{\Gamma}{8}(Re - 1000)Pr}{1 + 12.7\sqrt{\frac{\Gamma}{8}}(Pr^{\frac{2}{3}} - 1)} \left(1 + \left[\frac{d}{l}\right]^{\frac{2}{3}}\right) \qquad (11.15)$$

This equation again reflects the transition area and the entirely turbulent flow range. $\Gamma$ can be computed from the Reynolds number with:

$$\Gamma = (1.82 \log Re - 1.64)^{-2} \qquad (11.16)$$

The heat transfer coefficient finally can be derived again via:

$$\alpha = Nu \frac{\lambda}{d} \qquad (11.17)$$

## 11.4 The Junction Model

The JunctionT1 model just serves to add the fluid flows from two inlets to one outlet flow. Neither specific pressure drop effects nor heat transfer effects are considered.

The model however in backiteration steps of the boundary problem DEQ solving also works in reverse order by assuming inlet flows from left and right side and by saving the ratio derived in the previous iteration.

## 11.5 The Filter Model

The FilterT1 model reflects the effects of a gas flow through a gas filter holding back any dust or other particles. The pressure drop of the gas flow is modeled by a linear dependency equation. The pressure drop at 0.0 ks/s flow is obviously 0.0 bar. The reference pressure drop at a given reference mass flow can be specified in the input file.

Heat transfer from filter housing to fluid is modeled via the same equations as for the PipeT1 model. The filter however is modeled as one block and is not divided into subsections as the pipes are.

## 11.6 The Pressure Regulator Model

The PRegT1 class implements a model for dome pressure regulators. The output pressure can be specified as polynomial dependent function of the input pressure via according coefficients in the inputfile. The gas throttling from input pressure level down to output pressure however again reflects real gas effects of the Helium fluid, i.e. the temperature increase during throttling is computed.

Throttling first is computed as being adiabatic (and thus isenthalpic) and heat transfer from regulator housing to pressure gas then is added in a subsequent step.

Concerning the throttling the model first computes the Joule-Kelvin coefficient $\mu$ of Helium corresponding to the input flow conditions. This computation has to be very precise since at least at start of stage operations the $\Delta p$ is rather significant and deviations in the JK-coefficient would lead to errorneous outlet temperatures. So the temperature after throttling can be derived as being:

$$T_{out} = T_{in} + \mu ( p_{in} - p_{out} ) \qquad (11.18)$$

The heat transfer is computed in analogy to the method used for the PipeT1 and gas FilterT1 models.

## 11.7 The Pipe Split Model

The SplitT1 model serves to split the fluid flow from one inlets to two outlet flows. Neither specific pressure drop effects nor heat transfer effects are considered.

The model in backiteration steps receives the mass flow boundary conditions requested from the components connected downstream the outlets and splits the inflow value in the according ratio – not caring whether the overall inflow rate is sufficient. To assure this the sum of the massflows requested at the outlets as boundary conditions is handed over to the next upstrem inlet component.

The overall iterative boundary condition solving method is described in [3].

## 11.8 The Propellant Tank Model

The combined fuel/oxidizer tank is modeled by means of the class TankT1. This model includes a complete implementation of the tank physics DEQ system which models the behaviour over time in both tank compartments. The thermodynamic effects modeled is limited to medium energetic fuel/oxidzer combinations, i.e. hydrazine derivates as fuel and nitrogen oxides as oxidizers. For high energetic fuel/oxidizer combinations additional physical effects of cryogenic fluids have to be considered [1].

Please note that when looking into the sourcecode lots of variables can be found with slightly cryptic names and 6 characters name length – all uppercase. This is a heritage from the former FORTRAN77 implementation of the tank pressurization system in [1]. The following recipe should be sufficient to trace down the variable meanings:

- The first 2-3 letters indicate the physical variable. E.g. a heat transfer coefficient Alpha is called Alxxxx.
- The middle part typically indicates e.g. whether the variable describes something about the outer wall (“Außenwand = AW) or the separation wall btw. Oxidizer and fuel tank (“Trennwand = TW). E.g. ALTWxx.
- The last letter(s) indicate whether the variable describes the tank wall section in contact with fuel (“Brennstoff” = B) or oxidizer (“Oxidator” = O) and whether the part with gas (G) or liquid (L) contact is described. E.g. ALTWGO is the variable for the heat transfer coefficient Alpha between the separation wall and the gas in the oxidizer compartment.

### 11.8.1 The Differential Equation System

The processes inside the tank which take place during rocket stage operation can be reflected by a system of coupled first order ordinary differential equations. The numerics problem to be solved is propagation all changes over time, i.e. an initial value problem. All variables describing the physics inside the tank shall be known at time $t$. For this point all derivatives of the to be integrated variables can be computed and the values for $t + dt$ are to be integrated numerically. The tank model therefore computes all variable derivatives by means of the method `DEQDeriv()` and provides them to the solver in a one dimensional array.

$$\dot{\overline{Y}} = \overline{F}(t, \overline{Y}) \qquad (11.19)$$

$t$ Start value of the independent variable time
$\overline{Y}$ Array. Solution of the DEQ system at time $t$

$$\begin{aligned} \dot{Y}(1) &= f_1(t, Y(1) \ldots . Y(n)) \\ &\vdots \qquad\qquad \vdots \\ \dot{Y}(n) &= f_n(t, Y(1) \ldots . Y(n)) \end{aligned} \qquad (11.20)$$

where $\dot{Y}(1)$ to $\dot{Y}(n)$ represent the values of the right side of the DEQs, i.e. the derivatives of the dependent variables. The integration of this DEQ system

$$\overline{Y}(t+dt)=\overline{Y}(t)+\int \dot{\overline{Y}}(t)\,dt \qquad (11.21)$$

is performed by a class called `DEqSys` which is part of the simulator kernel package, not the rocket propulsion library. This integrator also historically stems from [1] and uses a Runge-Kutta method of 4th /5th order taken over with minor adaptations from [5]. A detailed algorithm description can be found in this cited literature. The variable list of the DEQ system 11.21 is listed below:

| | |
|---|---|
| $t$ | Time |
| *Y(1)* | Mass of pressurization gas in oxidizer tank compartment |
| *Y(2)* | Temperature of gas phase in oxidizer tank compartment |
| *Y(3)* | Volume of gas phase in oxidizer tank compartment |
| *Y(4)* | Pressure in oxidizer tank compartment |
| *Y(5)* | Temperature of liquid oxidizer |
| *Y(6)* | Temperature of outer wall of oxidizer tank in contact with gas |
| *Y(7)* | Temperature of outer wall of oxidizer tank in contact with liquid |
| *Y(8)* | Temperature of separation wall of oxidizer tank in contact with gas |
| *Y(9)* | Temperature of separation wall of oxidizer tank in contact with liquid |
| *Y(10)* | Pressure in fuel tank |
| *Y(11)* | Mass of pressurization gas in fuel tank |
| *Y(12)* | Temperature of gas phase in fuel tank |
| *Y(13)* | Temperature of liquid in fuel tank |
| *Y(14)* | Volume of gas phase in fuel tank |
| *Y(15)* | Temperature of outer wall of fuel tank in contact with gas |
| *Y(16)* | Temperature of outer wall of fuel tank in contact with liquid |
| *Y(17)* | Temperature of separation wall of fuel tank in contact with gas |
| *Y(18)* | Temperature of separation wall of fuel tank in contact with liquid |
| *Y(19)* | Mass of liquid oxidizer in oxidizer tank compartment |
| *Y(20)* | Mass of liquid fuel in fuel tank compartment |

In analogy thereto the variables stored in field $\dot{\overline{Y}}$ of the TankT1 model represent the according derivatives. E.g.:

| | |
|---|---|
| $\dot{Y}(1)$ | Mass flow of pressurization gas in oxidizer compartment |
| $\dot{Y}(2)$ | Temperature gradient of gas phase in oxidizer compartment |
| $\dot{Y}(3)$ | Volume gradient of gas phase n oxidizer compartment |
| | etc. |

These derivatives $\dot{Y}$ *can be expressed as functions of the state variables* $Y$. The according thermodynamic equations are cited in the subsequent sections.

### 11.8.2 Thermodynamics of the Oxidizer Tank

Since the nitrogentetroxide as oxidizer has a non neglectable vapor pressure at 300K, the gas phase in the oxidizer compartment must be treated as mixture of both oxidizer vapor and pressurization gas Helium. Computations however have shown that the mass of gaseous oxidizer can be assumed to be constant over the rocket stage operations time [1]. The temperature in the tank decreases during propellant consumption which implies that oxidizer vapor would condense, but the overall volume of the gas phase increases. Thus despite the decreasing partial pressure of the oxidizer approximately the overall mass of oxidizer is kept in vapor state over the entire stage operations time – if extremely long coast phases are excluded.. This implies:

$$\dot{m}_{vap} = 0 \tag{11.22}$$

The time derivative of the intrinsic energy of the gas volume in the oxidizer compartment can be formulated as:

$$\dot{U} = \frac{d}{dt}\left(m_{vap}\, u_{vap} + m_{prgas}\, u_{prgas}\right) \tag{11.23}$$

From these two equations results:

$$\dot{U} = \dot{m}_{prgas}\, c_{v,\,prgas}\, T_{gas} + \left(m_{vap}\, c_{v,\,vap} + m_{prgas}\, c_{v,\,prgas}\right)\dot{T}_{gas} \tag{11.24}$$

The first law of thermodynamics formulated for the gas volume of the tank oxidizer compartment thus reads:

$$\dot{m}_{prgas}\, h_{prgas,in} + \dot{Q}_{gas} = \dot{m}_{prgas}\, c_{v,\,prgas}\, T_{gas} + \left(m_{vap}\, c_{v,\,vap} + m_{prgas}\, c_{v,\,prgas}\right)\dot{T}_{gas} + p_{gas}\, \dot{V}_{gas} \tag{11.25}$$

The generic gas equation rearranged to solve for the temperature is,

$$T_{gas} = \frac{V_{gas}\, p_{gas}}{R_{gas}\, m_{gas}} \tag{11.26}$$

with

$$m_{gas} = m_{vap} + m_{prgas} \tag{11.27}$$

The differentiation of equation 11.26 results in

$$\dot{T}_{gas} = \frac{\dot{V}_{gas}\, p_{gas}}{R_{gas}\, m_{gas}} - \frac{p_{gas}\, V_{gas}\, \dot{R}_{gas}}{R_{gas}^2\, m_{gas}} - \frac{p_{gas}\, V_{gas}\, \dot{m}_{prgas}}{R_{gas}\, m_{gas}^2} \tag{11.28}$$

thereby keeping $p_{gas}$ constant via pressure control. In this formula $R_{gas}$ represents the specific gas constant of the mixtrue from oxidizer vapor and pressurization gas. $R_{gas}$ can be computed from:

$$R_{gas} = \frac{m_{prgas} R_{prgas} + m_{vap} R_{vap}}{m_{gas}} \tag{11.29}$$

Considering equation 11.22 the differentiation then leads to:

$$\dot{R}_{gas} = \frac{\dot{m}_{prgas} R_{prgas}}{m_{gas}} - \frac{(m_{prgas} R_{prgas} + m_{vap} R_{vap}) \dot{m}_{prgas}}{m_{gas}^2} \tag{11.30}$$

The volume of the gas phase continuously increases during engine operation as oxidizer fuid is consumed. Thereby two effects are to be considered. One is the volume increase through direct oxidizer consumption by the engine, i.e. reduction of mass of liquid oxidizer in the compartment. The other effect is the change of oxidizer fluid density due to cold pressurization gas flowing into the tank and the gas phase in the tank cooling down accordingly and thus cooling the liquid oxidizer. Thus the volume change can be formulated as:

$$\dot{V}_{gas} = \dot{V}_{fl,out} + V_{fl} \frac{\dot{\rho}_{fl}}{\rho_{fl}} \tag{11.31}$$

The density of the liquid oxidizer can be approximated by a polynomial expression like

$$\rho_{fl} = C + D T_{fl} + E T_{fl}^2 \tag{11.32}$$

with C, D and E being constants. Thus the density change results as:

$$\dot{\rho}_{fl} = \dot{T}_{fl} (D + 2 E T_{fl}) \tag{11.33}$$

The first law of thermodynamics for the liquid phase volume in the oxidizer compartment of the tank can be expressed as:

$$\dot{T}_{fl} = \frac{\dot{Q}_{fl}}{m_{fl} c_{fl}} \tag{11.34}$$

With the equations 11.19, 11.28, 11.30, 11.31 and 11.34 thus 5 equations are available for computation of the 5 variables:

$$\dot{m}_{prgas}, \dot{T}_{gas}, \dot{R}_{gas}, \dot{V}_{gas}, \dot{T}_{fl}$$

These can be brought into the form:

$$\begin{aligned}\dot{m}_{prgas} &= F(m_{prgas}, T_{gas}, V_{gas}, T_{fl} \ldots) \\ \dot{T}_{gas} &= F(m_{prgas}, T_{gas}, V_{gas}, T_{fl} \ldots) \\ \dot{T}_{fl} &= F(\ldots)\end{aligned} \qquad (11.35)$$

It has to be noted that the heat fluxes cited in the above equations are sum-ups from several effects. E.g. the heat flux imposed onto the gas phase in the tank adds up from heat transferred to the gas phase from the outer tank wall, from the separation wall and from the warmer liquid oxidizer phase.

The required heat exchange coefficients are computed via Nußelt equations. Thereby the equation for free convection in spherical vessels is used for the heat transfer from the tank walls to the gas phase (same as in high pressure bottle model):

$$Nu_d = 0.098 (Gr_d Pr)^{0.345} \qquad (11.36)$$

For the heat transfer between liquid and gas phase the heat transfer equation for a horizontal plate being heated from below is applied:

$$Nu = 0.14 (Gr_d Pr)^{0.333} \qquad (11.37)$$

The tank object distinguishes between outer and separation wall since the separation wall is modeled as 2 layers with half of the real material thickness each. One transferring heat into the oxidizer compartment and one into the fuel compartment. To avoid a coupling of the oxidizer and fuel compartment DEQ systems and risking the overall DEQ set to become stiff (see [3]) no heat transfer cross the separation wall is considered. Since the temperature ranges in both compartments anyhow are quite close to each other [1] the resulting error definitely can be neglected.

Finally the equations for the control volume tank wall itself have to be set up. Here the convention is that heat transfers from wall to gaseous / liquid fluid are counted as positive values. For any wall (outer or separation) and for the part in contact with the liquid phase the following equation applies:

$$\dot{T}_{Wall,fl} = -\frac{\dot{Q}_{Wall,fl}}{m_{Wall,fl} c_{Wall,fl}} \qquad (11.38)$$

And for the part in contact with gas phase:

$$\dot{T}_{Wall,gas} = -\frac{\dot{Q}_{Wall,gas}}{m_{Wall,gas} c_{Wall,gas}} + \frac{\dot{m}_{Wall,gas} c_{Wall} (T_{Wall,fl} - T_{Wall,gas})}{m_{Wall,gas} c_{Wall,gas}} \qquad (11.39)$$

The second term in addition considers the wall mass which comes into contact with the gas per second due to sinking liquid level in the tank compartment.

### 11.8.3 Thermodynamics of the Fuel Tank

The equation system for the fuel compartment of the tank is similar to that for the oxidzer compartment. For the fuel however the gas phase can be approximated as to consist of pure pressurization gas, since the vapor pressure of MMH or similar fuels is considerably low. The derivative of the intrinsic energy of the gas phase thus is reflected by:

$$\dot{U}_{gas} = \dot{m}_{prgas} u_{gas} + \dot{u}_{gas} m_{gas} \qquad (11.40)$$

With:

$$u_{gas} = c_{v,gas} T_{gas} \qquad \dot{u}_{gas} = c_{v,prgas} \dot{T}_{gas}$$

Even if the gas phase only is consisting of pressurant, for the terms describing gas inflow (pressure $p$, volume $V$, temperature $T$ and mass $m$) the index $prgas$ is used and for the variables describing the state inside the tank the index $gas$ is used. The same applies for material properties such as gas constant.

So the first law of thermodynamics for the gas phase of the fuel compartment is reflected by:

$$\dot{m}_{prgas} h_{prgas,in} + \dot{Q}_{gas} = \dot{m}_{prgas} c_{v,prgas} T_{gas} + (m_{gas} c_{v,gas}) \dot{T}_{gas} + p_{gas} \dot{V}_{gas} \qquad (11.41)$$

From the generic gas equation

$$m_{gas} = \frac{p_{gas} V_{gas}}{R_{gas} T_{gas}} \qquad (11.42)$$

via transformation and derivation the temperature gradient results as:

$$\dot{T}_{gas} = -\frac{\dot{m}_{gas} T_{gas}}{m_{gas}} + \frac{T_{gas} \dot{V}_{gas}}{V_{gas}} \qquad (11.43)$$

For the volume increase of the gas phase in the fuel tank the same equation applies as for the oxidizer compartment:

$$\dot{V}_{gas} = \dot{V}_{fl,out} + V_{fl} \frac{\dot{\rho}_{fl}}{\rho_{fl}} \qquad (11.44)$$

And in a similar approach as for the oxidizer the densitiy of liquid fuel as function of temperature can be approximated by [1]:

$$\rho_{fl} = C + D\,T_{fl} \tag{11.45}$$

*C* and *D* again are constants. The density gradient thus can be expressed by:

$$\dot{\rho}_{fl} = D\,\dot{T}_{fl} \tag{11.46}$$

The first law for the liquid volume reduces to the same equation as 11.34. The heat transfer coefficients again are resulting from 11.36 and 11.37. For the temperature gradients of the tank walls again the formulas 11.38 and 11.39 apply. Therewith also for the fuel tank a complete equation set is available which can be rearranged to the form of equation 11.35.

### 11.8.4 Thermodynamics of the Tanks in Blowdown Mode

The tank model supports computing the tank physics in blowdown mode when no more pressurization gas is fed in from the high pressure bottles due to closed valves and the rocket stage just is operated until the overall tank pressure has reduced to a certain minimum needed for engine feed.

Since the gas phases now do not reflect open but closed control volumes now, the first law of thermodynamics changes to slightly different equation:

$$\dot{U} = \dot{Q}_{gas} - p_{gas}\,\dot{V}_{gas} \tag{11.47}$$

Via rearrangement and insertion of $\dot{U} = m\,c_v\dot{T} + \dot{m}\,c_v T$ and $\dot{m} = 0$ it results to be:

$$\dot{T}_{gas} = \frac{\dot{Q}_{gas}}{m_{gas}\,c_{v,gas}} - \frac{p_{gas}\,\dot{V}_{gas}}{m_{gas}\,c_{v,gas}} \tag{11.48}$$

In this equation for the oxidizer compartment for $c_{v,gas}$ the value of the gas mixture from oxidizer vapor and pressurization gas is to be used. For the fuel compartment the value of the pure pressurization gas.

The pressure gradient in the tank results from the differentiated generic gas equation:

$$\dot{p}_{gas} = \frac{m_{gas}\,R_{gas}\,\dot{T}_{gas}}{V_{gas}} - \frac{m_{gas}\,R_{gas}\,\dot{V}_{gas}}{V_{gas}^2} \tag{11.49}$$

Also here for the specific gas constant $R$ here in case of the oxidizer compartment the

value for the mixture is to be applied and for the fuel compartment the value for the pure pressurization gas.

In blowdown mode the required derivatives to be computed are:

$$\dot{T}_{gas}, \dot{p}_{gas}, \dot{V}_{gas}, \dot{T}_{fl}$$

With the equations 11.48, 11.49, 11.31 and 11.34 as well as with the formulas for the heat transfers 11.36, 11.37, 11.38 and 11.39 for each tank compartment again a complete equation set is available which can be rearranges into the form of equations 11.35

## 11.9 The Flow Valve Model

The FluidFlowValve is a simplistic representation of a flow control valve. It has a reference mass flow and a reference pressure loss and an analog control line port. Depending on the analog signal on the control line (between 0.0 and 1.0) the valve sets the fluid flow and pressure loss as linear interpolation accordingly. Non-linearities are not considered yet.

## 11.10 The EngineController Model

The EngineController is a unit designed for control of oxidizer and fuerl flows to the engine of the modelled rocket. It provides two analog line outputs which e.g. can be connected to FluidFlowValve components.

In the Engine Controller for each output line the function of the control value over time has to be programmed by the user according to the user's needs. The delivered example code starts with a signal value read from input file and reduces the output signal within 200sec to zero.

## 11.11 The IntervalController Model

The IntervController also is a unit designed for control of oxidizer and fuerl flows to the engine of the modelled rocket. It provides two analog line outputs which e.g. can be connected to FluidFlowValve components.

This model class allows to specify up to 10 time intervals with individual control output value setting. The start / end time and value for each interval are loaded from the according input file section – for both of the output lines.

Refering to Figure 6 the IntervalController and the EngineController can alternatively be used to control the rocket stage engine.

## 11.12 The Engine Model

### 11.12.1 Basics

In previous versions of OSK the Engine thrust was only calculated with a constant ISP value with

$$F = \dot{m}\, ISP\, g_0 \qquad (11.50)$$

where $\dot{m}$ is the total mass flow of oxidizer and fuel into the thrust chamber.

To get a more realistic behavior, the ISP is calculated from

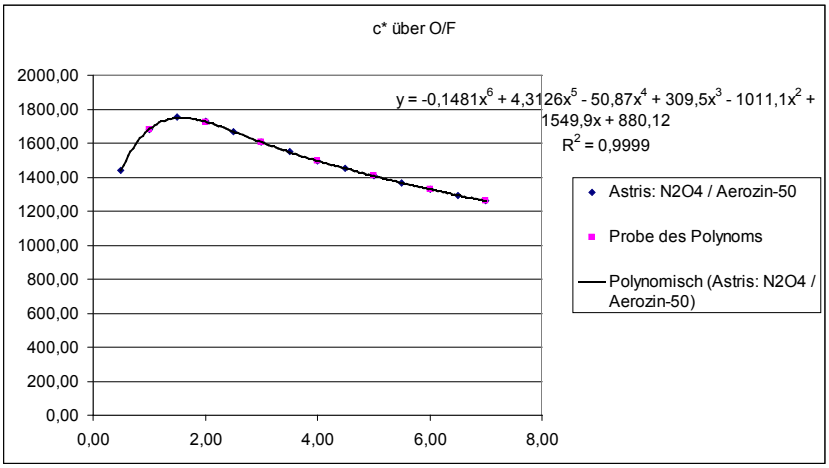
$ISP = \frac{c * cf}{g_0}$ where c* [m/s] is the characteristic velocity of the combustion gases in the thrust chamber. c* is mainly a function of the mixture ratio $OF = \frac{\dot{m}_{ox}}{\dot{m}_f}$ of oxidizer to fuel mass flow rate. Now, the nonlinear behavior of the thrust can be calculated with:

$$F = \dot{m}\, c * c_f \qquad (11.51)$$

The values for c* and cf are theoretical values with 100% efficiency, thus efficiency factors for c* and $c_f$ are used. $c_f$ describes the thrust factor, a number that describes the the augmentation of thrust from the thrust chamber in the nozzle. The value of $c_f$ depends mainly on the ambient pressure and the conditions in the thrust chamber. The altitude for the ambient pressure calculation is currently static and needs to be imported from the Lat-Long-Alt routine.

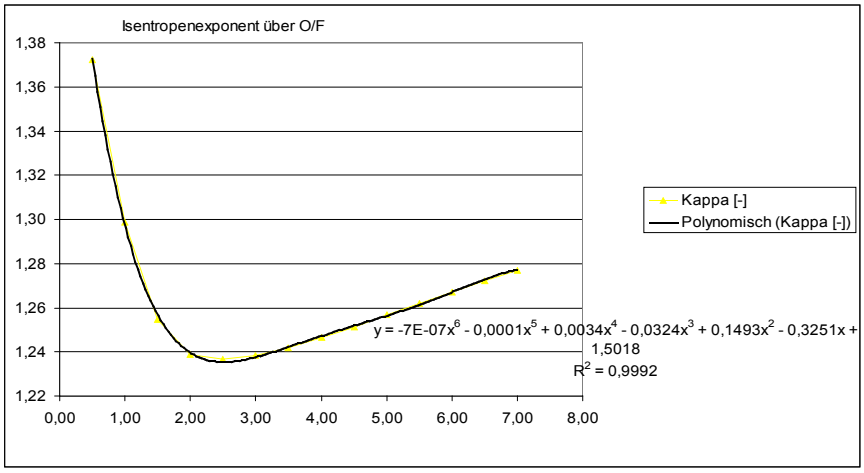
#### *11.12.1.1 Characteristic Velocity c**

A polynomial for c* has been calculated with NASA's CEA Gordon McBride ( http://www.grc.nasa.gov/WWW/CEAWeb/ ).



The values for c* from the polynomial are ideal without losses, so a factor for the combustion efficiency $\eta$ ·c* has been introduced,with a given value of 0.94.

The isentropic exponent or ratio of specific heats kappa ( $\kappa$ ) of the combustion gases is also a function of OF and has also been programmed as a polynomial from CEA ( picture below )




### *11.12.1.2 Thrust Factor $c_f$*

The effect of altitude adaption of the nozzle is integrated with the thrust factor $c_f$. To provide the ambient pressure a simple atmospheric model has been included ( http://www.grc.nasa.gov/WWW/K-12/airplane/atmosmet.html ). With the formula for the area ratio of the nozzle,

$$\epsilon = \frac{A_e}{A_t} = \Gamma \left(\frac{p_e}{p_c}\right)^{\left(\frac{-1}{k}\right)} \left\{\frac{2k}{k-1}\left[1-\left(\frac{p_e}{p_c}\right)^{\left(\frac{k-1}{k}\right)}\right]\right\}^{(-0.5)} \qquad (11.52)$$

with

$$\Gamma = \sqrt{k\left(\frac{2k}{k+1}\right)^{\frac{k+1}{k-1}}} \qquad (11.53)$$

which assumes 1D isentropic adiabatic flow, the nozzle exit pressure $p_e$ is calculated iteratively with the Newton-method. With the exit pressure $p_e$, ambient pressure $p_a$ and chamber pressure $p_c$ the thrust factor $c_f$ can be calculated for each time step.

$$c_f = k^{0.5}\left(\frac{2}{k+1}\right)^{\frac{k+1}{2(k-1)}} \left\{\frac{2k}{k-1}\left[1-\left(\frac{p_e}{p_c}\right)^{\left(\frac{k-1}{k}\right)}\right]\right\}^{(-0.5)} + \frac{A_e}{A_t}\frac{(p_e-p_a)}{p_c} \qquad (11.54)$$

If the calculated nozzle exit pressure is to low, flow separation can occur. Then the value for the thrust may no more valid. The criterion of Summerfield is used to check for flow separation in the nozzle: $p_e < 0.4 * p_a$
In such case a LOG.debug message is created as warning by the model.
Because of the ideal 1D flow with losses, an efficiency factor for $c_f$ is used with a constant value of 0.99.

## 11.12.2 Model Verification

### *11.12.2.1 Verification of c**

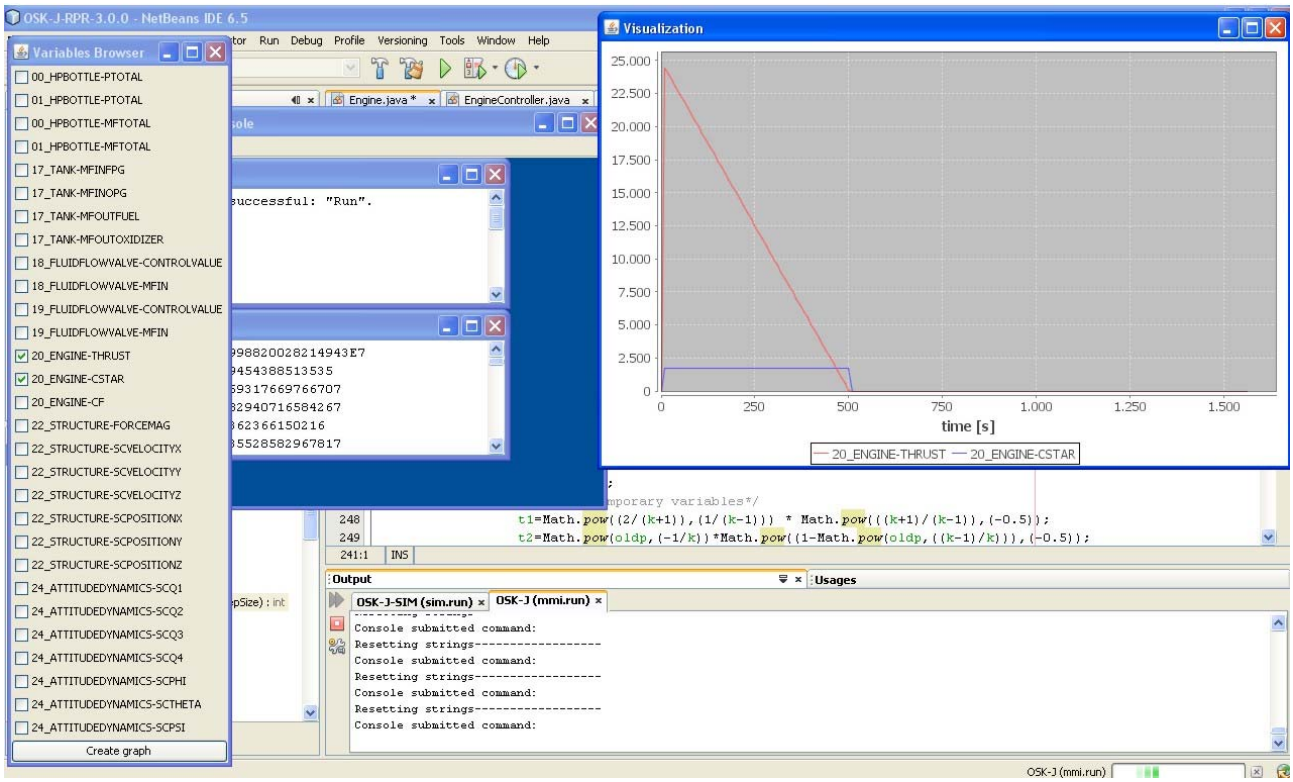



*Figure 10: Engine model behavior during throttling with ideal oxidizer/fuel mix ratio.*

A first test has been done by letting the engine controller reduce the oxidizer and fuel mass flow and therefore the thrust linearly from the maximum value at start to zero. Therefore the oxidizer to fuel ratio stays constant and the c* should also be constant as it can be seen in the above picture.

To show the nonlinear behavior of the engine with changing oxidizer to fuel ratio, the engine controller has been set to only reduce the fuel mass flow. So the c* is nonlinear decreasing from its maximum value at start with optimum oxidizer to fuel ratio, to lower values as it can be seen in the following picture. The nonlinear decay can also be seen in the thrust diagram.

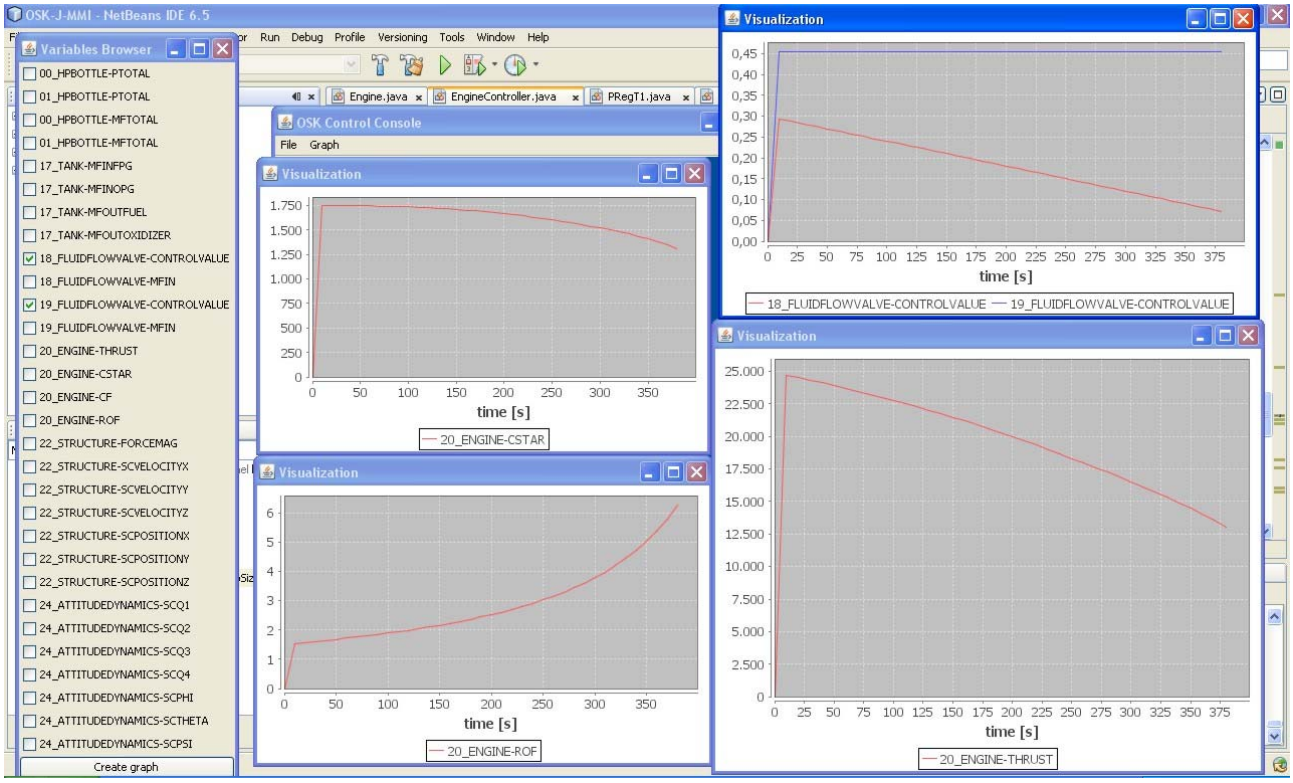



*Figure 11: Engine model reflecting non-ideal oxidizer/fluel mix ratios.*

#### *11.12.2.2 Verification of $c_f$*

The current altitude of the simulation orbit is 600km, where the ambient pressure is almost zero. So no effects of ambient pressure to the thrust coefficient $c_f$ can be noticed. The value and change of $c_f$ becomes important in the lower altitudes, especially for first and second stages of launch vehicles. For verification of these effects the orbit has been changed to 30km to see the effects of $c_f$.

For a simulated altitude of 30km, $cf$ was 1.8, ISP=300s and thrust=22200 N. Whereas for 100km, $c_f$ was 1.875, ISP=310.99 with a thrust of 23000 N. This value is pretty close to the literature value ( http://www.astronautix.com/engines/astris.htm ), which says the vacuum ISP is 310 [s] and thrust is 23000 N for the Astris engine.

## 11.13 The Structure Model

The structure is modeled as a point mass with the according forces (engine thrust, gravity acceleration) imposed to it. The according S/C overall acceleration vector, the velocity and position are integrated in ECI coordinates applying 4$^{th}$ order Runge-Kutta-Integrators.

The velocity and position are converted from ECI to ECEF coordinate system by applying source code which originally was implemeted as Matlab code at the Institute of Space Systems. This code considers rotation, precession and nutation which are depicted in Figure 12. Polar motion is disregarded because its effect is minor and can furthermore not be predicted correctly.

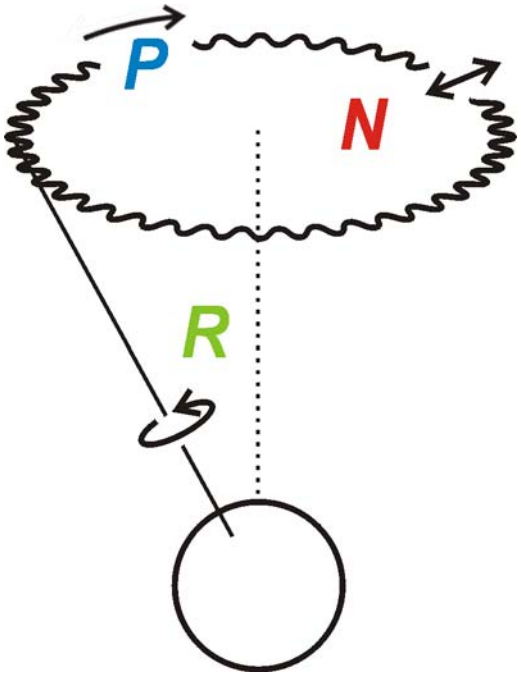



*Figure 12: Earth's motion.*

In a first step, the UTC time which was committed to the method is converted both to UT1 and to TT. Time is needed in these formats for further calculations. Afterwards the Greenwich Mean Sidereal Time is calculated by applying

$$GMST = GMST(J2000) + \omega_{Earth} \cdot 86400 \cdot UT1_{J2000} \qquad (11.55)$$

which refers to the Greenwich Mean Sidereal Time in J2000. Consequently, the transformation matrix comprising rotation effects can be calculated:

$$\Theta = \begin{pmatrix} \cos(GMST) & \sin(GMST) & 0 \\ -\sin(GMST) & \cos(GMST) & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad (11.56)$$

In the next step specific angles are calculated by applying correlations. With these angles and trigonometric functions, the transformation matrix P comprising precession effects is determined.

In the next step the longitude of the ascending node of the moon is calculated because it has an enormous impact on the Earth's nutation. Therefrom both periodical change of the position of the Earth's vernal equinox and periodical change of the obliquity of the ecliptic can be determined. Together with the obliquity of the ecliptic at J2000 and trigonometric functions, the transformation matrix N comprising nutation effects is calculated.

Finally the matrices comprising different effects can be summarized to one transformation matrix U:

$$U = \Theta \cdot N \cdot P \tag{11.57}$$

Thus, the ECEF position can be obtained with

$$R_{ECEF} = U \cdot R_{ECI} \tag{11.58}$$

As the velocity is the derivative of the position, it can be calculated applying the product rule

$$V_{ECEF} = \dot{R}_{ECEF} = U \cdot V_{ECI} + \dot{U} \cdot R_{ECI} \tag{11.59}$$

with

$$\dot{U} = \frac{d(\Theta \cdot N \cdot P)}{dt} = \frac{d\Theta}{dt} \cdot N \cdot P \tag{11.60}$$

# Part III

# 12 The *OpenSimKit* Guide Through Galaxy

As cited already in section 2 *OpenSimKit* provides an interface to Celestia [6] for visualizing simulated spacecraft in Orbit in 3D. Setting up this interface requires a handful of manual steps but it is manageable for people like Arthur Dent, so:

**Don't panic !**

Douglas Adams [8]

## 12.1 Introduction to Celestia

Celestia is an astrodynamics software which models the entire solar system with all planets, the moons, comets, major asteroids etc. It allows the user to place his viewpoint at an arbitrary position in the solar system and watch bodies moving. The user may accelerate simulated time or slow simulation down. Celestia also allows visualization of man-made objects, such as the International Space Station or other famous spacecraft.

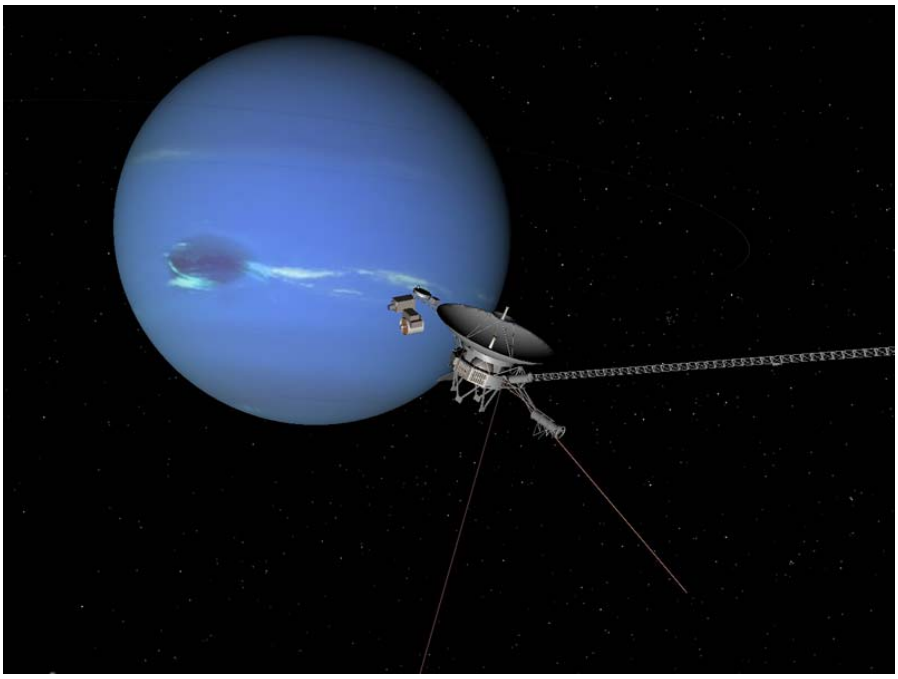


*Figure 13: Celestia scene: Voyager 2 passing Neptune*

This feature is used for visualizing objects simulated by *OpenSimKit*. Especially for this purpose the Astris rocket upper stage geometric model was established.

## 12.2 Concept of Interfacing Celestia from *OpenSimKit*

Interfacing Celestia from *OpenSimKit* is done by periodically supplying Celestia about position and attitude data of the spacecraft as being computed in *OpenSimKit*. This is done by writing the position and attitude information onto a socket in *OpenSimKit* and

making Celestia read from the socket. Subsequently the spacecraft geometry is visualized by Celestia at scripted orbit position and in scripted attitude including proper illumination depending on sun/eclipse position etc.

By default Celestia has fixed formulas for computing body movements along orbits and is not prepared to be supplied with position / attitude from external. However since Celestia V1.5.1 it provides the feature of writing extension plugins in a scripting language called Lua [9]. And Lua provides a socket interface library and thus permits interfacing to the *OpenSimKit* simulator. This feature is called “scripted attitude” respectively “scripted orbit position” in the Celestia documentation.

To realize the visualization of the included Astris rocket model as *OpenSimKit* calculates its position and attitude, the following is required:

- Installed *OpenSimKit* V3.5.0 or higher
- Celestia 1.5.1 or higher – preferably 1.6.0 or higher
- Lua 5.1 or higher

- The Lua socket library 2.0.2 or higher [11]
- And a geometry model of the simulated spacecraft – e.g. the Astris rocket stage
- and some Lua routines and Celestia configuration files to bring everything together.

All components are available for both Linux and Windows platforms. The setup described in the subsequent paragraphs works for both platforms. Differences only appear considering the install directories of Celestia and Lua.

## 12.3 Steps for the Infrastructure Setup

**Installing *OpenSimKit*:**
At first a Celestia compliant *OpenSimKit* release has o be installed, i.e. V3.5.0 or higher. Please refer to chapter 5 for the according steps.

**Installing Celestia:**
On a lot of Linux platforms, Celestia is already installed by default and no installation tasks have to be performed explicitly. For the popular Linux implementations such as SuSE or Ubuntu or RedHat precompiled packages are available for installation and do not require Celestia builds from sourcecode. The same applies for all MS Windows Platforms from XP onwards.
In case your operating system does not include a preinstalled Celestia, the Celestia homepage and installation root can be found in [7]. Just follow the instructions to receive the standard installation which nowadays is V1.6.0.

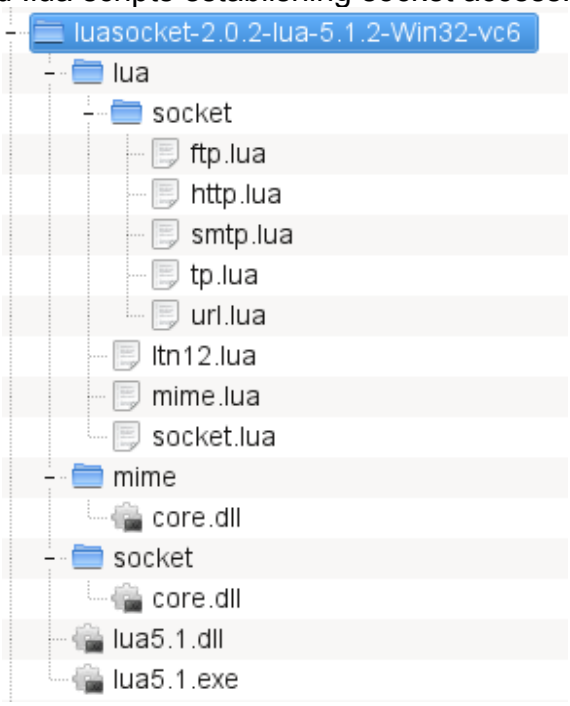
**Installing Lua:**
Lua by default is pnot reinstalled on Windows nor on Linux platforms. While on Windows platforms you have to install it from the Lua root [10], on most Linux-es there are packages

for Lua included in the distributions (e.g. Ubuntu, SuSE). Use the according installation tool and install Lua (via apt-get, YaST or similar). Make sure you install version 5.1 or higher.

Now come the bit more clumsy steps, but this recipe should navigate the user properly through the jungle even as a beginner.

**Installing the Lua socket library - Windows:**

Under Windows XP for Lua 5.1 or higher, the luasocket library from [12] is already included in the Lua standard installation described in the paragraph above. This comprises the according .dll files and .lua scripts establishing socket access:




No additional steps are required for installation.

**Installing the Lua socket library - Linux:**

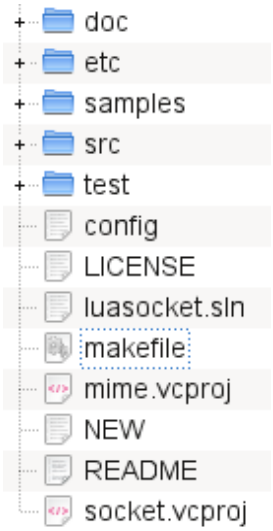
For some Linuxes – such as SuSE 11.2 - the socket library is not included by default in the Lua installation rpm. So the luasocket library has to be downloaded from [12] and manual installation is required. The .tar.gz archive has to be downloaded and stored into a convenient work directory - e.g.:

```
/luainstallwork
```

Unpacking the archive will lead to a directory structure such as

```
/luainstallwork/luasocket-2.0.2/
```

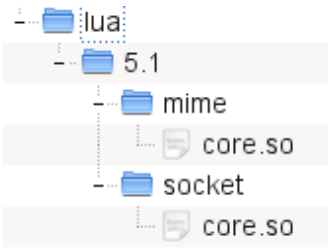
For other Linux variants – such as SuSE 11.3 – a precompiled RPM packet is available already from the Linux supplier's repositories. In such case this luasocket package just is to be installed by the standard package manager of the Linux distribution and the rest of this paragraph can be skipped.




Go into the directory with a text console and initiate the `make` command. This will compile some c-sources from the ./src subdirectory and install the binaries as well as a number of .lua files from the ./etc directory onto your computer. Having fulfilled these tasks you are ready to run the application. However, some additional steps are provided below to verify that the Lua sockets are properly installed and working.

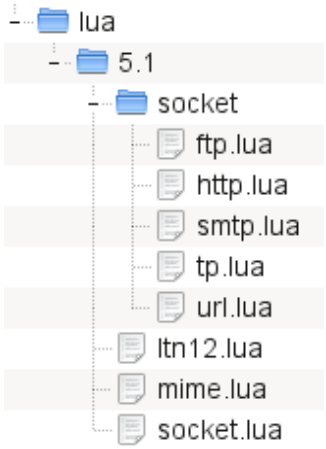
First the proper installation of the lua libraries into the relevant directories can be checked. Please note that on Linux the target hardware dependent files are stored in a different subdirectory than the target hardware independent ones. So there should be a folder structure like:

`/usr/local/lib/`




For the target hardware-independent files the structure should look like:

`/usr/local/share/`



The Linux download comes with several example files. Two of them can be used for an intuitive and simple test. To verify that the sockets are properly working, start two text consoles and navigate to

```
/luainstallwork/luasocket-2.0.2/samples
```

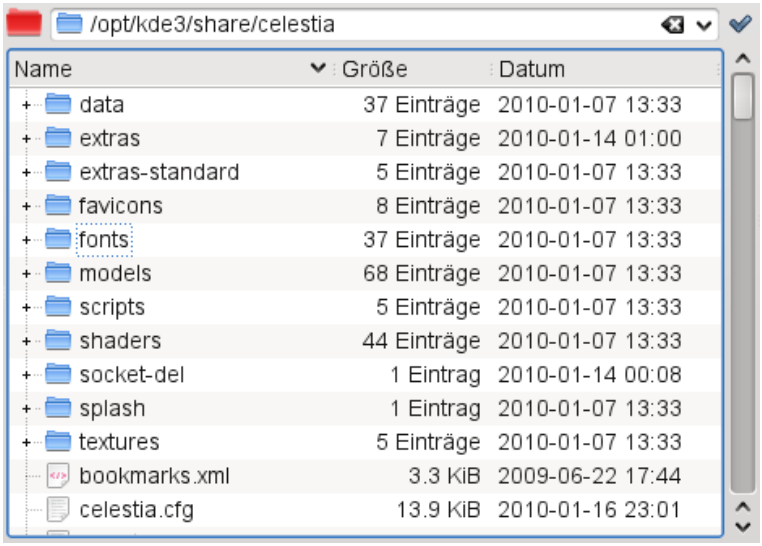
There you find a “listener.lua” and a “talker.lua” program. Start them (listener first) and all you type into the listener will be echoed by the talker as soon as you press <CR>.

```
> lua talker.lua
Attempting connection to host
'localhost' and port 8080...
Connected! Please type stuff (empty
line to stop):
whow, it seems to work
```

```
> lua listener.lua
Binding to host '*' and port 8080...
Waiting connection from talker on
0.0.0.0:8080...
Connected. Here is the stuff:
whow, it seems to work
```

**Configuring Celestia to visualize the *OpenSimKit* Rocket Stage:**

After Celestia, Lua and the socketlib are installed, Celestia must be customized manually to visualize the *OpenSimKit* Rocket.


/opt/kde3/share/celestia
Name
Größe
Datum
data 37 Einträge 2010-01-07 13:33
extras 7 Einträge 2010-01-14 01:00
extras-standard 5 Einträge 2010-01-07 13:33
favicons 8 Einträge 2010-01-07 13:33
fonts 37 Einträge 2010-01-07 13:33
models 68 Einträge 2010-01-07 13:33
scripts 5 Einträge 2010-01-07 13:33
shaders 44 Einträge 2010-01-07 13:33
socket-del 1 Eintrag 2010-01-14 00:08
splash 1 Eintrag 2010-01-07 13:33
textures 5 Einträge 2010-01-07 13:33
bookmarks.xml 3.3 KiB 2009-06-22 17:44
celestia.cfg 13.9 KiB 2010-01-16 23:01

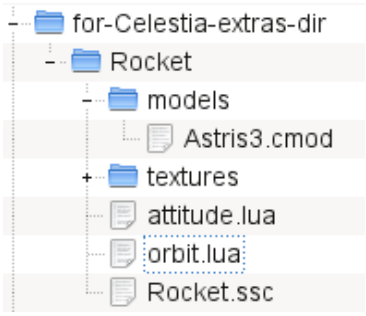
This comprises providing the Rocket geometry and configuring Celestia to invoke the relevant Lua routines. They read position and attitude data from the socket to which *OpenSimKit* transmits them.

For this purpose. a Celestia-Files package can be downloaded from the *OpenSimKit* webpage for *OpenSimKit* V3.5.0 or higher. This package contains all files required to configure Celestia. Please note that for Linux you'll need to activate your shell or filebrowser in administrators mode (sudo) to have the appropriate write permissions to the Celestia main directory.

The figure above shows the Celestia main directory as an example for a Linux / KDE system. There are a number of files plus the `./extras` subdirectory to be mentioned. The Celestia-Files package from the *OpenSimKit* webpage contains two subdirectories. One of them contains files to be placed inside the Celestia main directory. and the other one

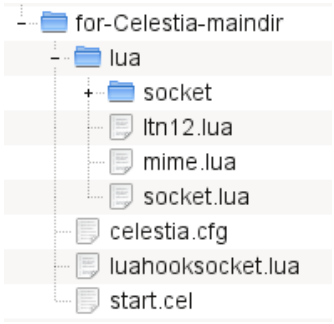
contains files to be put into the Celestia `./extras` subdirectory. File installation can be done just by copying the files from the *OpenSimKit* package into the according Celestia locations.

The `./for-Celestia-extras-dir` directory comprises all files concerning the Rocket model visualization. The substructure should appear as follows:




The main files to be mentioned are the `Rocket.ssc` telling Celestia that there is a spacecraft called “Rocket” visualized by a geometry 3d file called `Astris3.cmod` and lua files for orbit and attitude computation.

The `./for-Celestia-maindir` comprises all stuff to be placed in the min directory. The structure is as follows:




The figure shows the `./lua` directory which contains all the lua routines for the sockets again. It is a copy of the installation from the socket library since Celestia wouldn't be able to find them in the original installation location.

Another important file is the `start.cel`. It is a modification of the original `start.cel` script coming with the Celestia installation. This file contains the settings that at the start-up Celestia sets view and focus to the initial position of the Rocket. See lines 25-40 of the file for further details.

The delivered `celestia.cfg` is a modification of the original file. There is the new entry:

```
LuaHook "luahooksocket.lua"
```

It tells Celestia that there are Lua extension scripts to be used and that the main Lua routine is `luahooksocket.lua`. `luahooksocket.lua` generates the communication socket instance in Lua and connects to socket 1520 which *OpenSimKit* writes the spacecraft position and attitude information on. After the connection has been established, Celestia periodically reads the provided data to be used by the Rocket's `attitude.lua` and `orbit.lua`.

**Running *OpenSimKit* and Celestia:**

After the installationi s finished, *OpenSimKit* can be run with Celestia visualizing the Rocket. Start the *OpenSimKit* simulator and MMI as usual. As soon as the simulation is running, start Celestia.
Enjoy the magnificent view from *The Restaurant at the End of the Universe* onto the flying rocket stage ;-)

*Figure 14: Rocket stage over North America. Moon in the background.*

# Part IV

# 13 Developing Component Models for *OpenSimKit*

The preferred method is to use the Eclipse Java Development Tools (JDT). Create a new Java Project from existing source. Use the *OpenSimKit* directory in the source zip file as a base. The class "org.opensimkit.InteractiveMain" is the main entry point for OSK-J-SIM. The class "org.opensimkit.console.OSKConsole" is the main entry point for OSK-J-MMI.

## 13.1 Using Eclipse JDT 3.3.2

Go to "File|New Project|New Java Project". Enter a project name (obviously you could use "OSK-J" as project name). Under Contents check "Create project from existing source" and browse for the directory where you placed your copy of the OSK-J sources. Click "Next >" to continue.

## 13.2 Using Netbeans 6.1

Go to "File|New Project" in the following dialog choose "Java" from the categories on the left and "Java Project with Existing Sources" from the projects on the right. Enter a "Project Name", for example "OSK-J", and as "Project Folder" choose the one where you placed your copy of the OSK-J sources. The "Build Script Name" (build.xml) should be left alone. Click "Next >" to continue.

## 13.3 How to add a new Dependency to *OpenSimKit*

To be described.

# 14 Targets of the Ant build.xml file

To build selected applications of *OpenSimKit* instead of the entire *OpenSimKit* suite different distribution targets can be accessed individually. E.g. for building the simulator only after some code changes, the user can rebuild it by typing

```
ant sim.dist
```

into the terminal console - presuming the terminal root directory is the ./build subdirectory of the *OpenSimKit* distribution.

## 14.1 Global Targets for both the Simulator and the MMI:

`all`

This target generates the MMI and the simulator distribution as well as the sample fluid system component library. This includes the jar files as well as the javadoc documentation.

`clean`

This target cleans the temporary directories as well as the directories containing the jar files of the MMI, the simulator and the fluid system component library.

## 14.2 MMI Targets:

`mmi.dist`

Generates the MMI distribution. This includes the jar file as well as the javadoc documentation.

`mmi.build`

Creates the MMI jar file.

`mmi.javadoc`

Removes the old MMI javadoc documentation and creates a new one.

`mmi.javadoc.clean`

Cleans the javadoc documentation. It is implicitly called by the target "mmi.javadoc" to avoid having inconsistent HTML files.

`mmi.test`

Executes the MMI unit tests.

`mmi.clean`

Cleans the temporary directories as well as the directories containing the jar files of the MMI.

`mmi.run`

Executes the MMI application jar file.

## 14.3 Packet Library Targets:

`pkt.dist`

Generates the Packet Library. This includes the jar file as well as the javadoc documentation.

`pkt.build`

Creates the Packet Library jar file.

`pkt.javadoc`

Removes the old Packet Library javadoc documentation and creates a new one.

`pkt.javadoc.clean`

Cleans the javadoc documentation. It is implicitly called by the target "pkt.javadoc" to avoid having inconsistent html files.

`pkt.test`

Executes the Packet Library unit tests.

`pkt.clean`

Cleans the temporary directories and deletes the jar file of the Packet Library.

## 14.4 Rocket Propulsion System Targets:

`rpr.dist`

Generates the rocket propulsion system component library. This includes the jar file as well as the javadoc documentation.

`rpr.build`

Creates the rocket propulsion system component library jar file.

`rpr.javadoc`

Removes the old rocket propulsion system javadoc documentation and creates a new one.

`rpr.javadoc.clean`

Cleans the javadoc documentation. It is implicitly called by the target "rpr.javadoc" to avoid having inconsistent html files.

`rpr.test`

Executes the rocket propulsion system unit tests.

`rpr.clean`

Cleans the temporary directories and deletes the jar file of the rocket propulsion system.

## 14.5 Simulator Targets:

`sim.dist`

Generates the simulator distribution. This includes the jar files as well as the javadoc documentation.

`sim.build`

Creates the simulator jar files.

`sim.javadoc`

Removes the old simulator javadoc documentation and creates a new one.

`sim.javadoc.clean`

Cleans the javadoc documentation. It is implicitly called by the target "sim.javadoc" to avoid having inconsistent html files.

`sim.test`

Executes the simulator unit tests.

`sim.clean`

Cleans the temporary directories as well as the directories containing the jar files of the simulator.

`sim.run`

Executes the simulator application jar file.

# 15 *OpenSimKit* Architecture

This chapter describes the *OpenSimKit* architecture, the subsystems on which *OpenSimKit* is founded as well as the general intentions during the design of *OpenSimKit*.

## 15.1 Subsystem

*OpenSimKit* consists of the following subsystems:

- core
- commanding
- components
- manipulation
- ports
- steps
- xml

## 15.2 Model Libraries

Since release 2.5, *OpenSimKit* no longer bundles any simulation models directly in the `osk-j-sim.jar` file. Instead it supports a concept of so-called model libraries. Model libraries are simple jar files containing the compiled class files of any number of models as well as their dependencies. This is another step away from a “simple” simulator and towards a simulation framework. This separation of the simulator and its models allows users to simply share their model libraries and it makes *OpenSimKit* more flexible. The default *OpenSimKit* models of a rocket propulsion system are now included in the file `osk-j-rocket-propulsion.jar` in the “models” subdirectory of the *OpenSimKit* distribution. Those model libraries can be developed (almost) independent from the simulator. They only need the `osk-j-sim.jar` as compile-time dependency, because the latter include the interfaces and base classes, which are needed to create an *OpenSimKit* model.

During start-up the simulator is looking for jar files inside the models directory. Any jar file found is included in the simulator's classpath. This is done by the method *addModelLibrariesToClasspath()* of the Class *InteractiveMain*. During the initialization of the simulation when the XML input file is read, the Classloader is now able to find the model classes and can load them into the Java Virtual Machine.

## 15.3 Packet Library

*Note:*

The following section describes work-in-progress, which means that in case of differences between the documentation and the source, the source is true. Additionally not all concepts described here are already fully implemented and tested!

The Packet Library is the core of the communication of the MMI and simulator. The Packet Library consists of two parts. The interfaces in the package org.opensimkit.packet and one sample implementation (named OSKPacket) in the package org.opensimkit.oskpacket. This separation is helpful to separate the interface from the implementation. If the *OpenSimKit* kernel only uses the interfaces then it is possible to exchange the current implementation, which is called OSKPacket, by a different - yet compatible – one.

## 15.4 OSKPacket specification

After reading this section, please take also a look at the Packet Library javadoc.

An OSKPacket is a binary chunk of data which contains either information from the simulator to the MMI (this is called telemetry (TM)) or information from the MMI to the simulator (this is called telecommand (TC)). An OSKPacket has the following structure(one octet is defined to be eight bits):

| **Offset** (octets) | **Size** (octets) | **Name** | **Remarks** |
|---|---|---|---|
| 0 | 1 | Packet type | Identifies a packet as telemetry or telecommand packet. |
| 1 | 1 | Application ID | Used to address different applications. 256 different applications can be addressed. |
| 2 | 1 | Sequence counter | A sequence counter to have a facility to detect lost packets. Each application shall have its own sequence counter. |
| 4 | 2 | Packet length | The packet length including header and footer (checksum). It can be equal to MAX_PACKET_SIZE at most. |
| 6 | 8 | Simulated time | The time of the simulation. This is in the default Java time format, which is Unix time with millisecond resolution. |
| 14 | 8 | System time | The time of the system. Normally this is local time. This is in the default Java time format, which is Unix time with millisecond resolution. |
| 22 | n | Data | The data size ranges from 0 to a maximum of 2027 octets. |
| 22+n | 2 | Checksum | The CRC16-CCITT checksum of the header and data. |

Basically it is split into three parts: header, data, footer. The header has a length of 22 octets and contains the most basic information for the communication between the MMI and the simulator. The packet type is used to distinguish TM from TC packets.

Note:
Currently only TM packets are used inside the simulator. The commands from the MMI to the simulator are still simple strings. This will change in future *OpenSimKit* releases.

The application ID is used to allow the definition of different kinds of packets which have different data inside. Currently it is only possible to define 256 different TM and 256 different TC packets.

Note:
It is proposed that an additional field called context should be added to allow more different packets to be defined. But this change will be discussed in the forum.
The sequence counter is used to track the reception of packets. It shall be incremented by one for every packet send. Each application ID shall have its own sequence counter. If the maximum value of the sequence counter is reached it shall overflow to 0.

Note:
Currently the sequence counter is not used in OSK and I am not sure how to deal with the inability of Java to have unsigned types. This means currently the sequence counter would be between -128 and 127 instead of 0 and 255.

The packet length tells the length of the packet including header and footer.
The simulated time is used to track the time inside the simulation. This is necessary as usually the simulation time is not the same as the local time. For easier handling with the Java language it is 8 byte long.

The system time is used to track the local time. This is necessary as usually the simulation time is not the same as the local time. For easier handling with the Java language it is 8 byte long.

The second part contains the user data. The definition of the data inside the packet and its position shall be defined beforehand by the sender and receiver. For each pair of “packet type” and “application ID” there shall at most be only one definition of the user data!

Note:
Realistically we need a database (could be Apache Derby based or “plain” XML) to easily define the data inside the packets. Because now everything must be hard-coded inside the MMI and the simulator. Any volunteers?

The third part called footer contains the check sum. Calculated from position 0 to position (22 + n) -2. In other words from the beginning of the packet to the end of the user data and thus omitting the footer.

# 16 Developing Equipment Models in *OpenSimKit*

This chapter describes the creation and use of a sample *OpenSimKit* system equipment model. For a first look at the *OpenSimKit* architecture a simple temperature sensor for the example rocket stage system is implemented.

It starts with the creation of a Java class representing the component model and goes on to explain the compilation process of the model. After this the input file is changed so that the *model* is actually used by the Simulator. After a simulation run the simulator created output file is analyzed.

## 16.1 Prerequisites

- Netbeans IDE (http://www.netbeans.org/); get at least Version 6.1.
- Create a Netbeans Free-form Project for the *OpenSimKit* Simulator as described in the document *Netbeans 6.x Integration.odt* found in the *osk-j-x.y.z-src/doc* directory.

As this tutorial focuses on the creation of a simulator component model it requires the use of the Netbeans IDE. Although it is possible to use another IDE or no IDE at all to create an *OpenSimKit* component model the constraint of using only one IDE makes this tutorial much more focused on model development instead of IDE differences. It is possible to use Netbeans 6.x for *OpenSimKit* Component development.

## 16.2 Writing a Java Class

For this tutorial it is best to include the new model in the rocket propulsion model library. The reason for this is that there exists already a build file for this library and this model can be easily seen in action when integrated into the default simulation. At first a new Java class is created with the name TemperatureSensor in the package org.opensimkit.models. If an IDE is used for development it is necessary to make sure that the new Java source file is located in the package org.opensimkit.models. If a plain text editor is used the new Java source file needs to be located in the directory “rpr\src\org\opensimkit\models”. To make this class an *OpenSimKit* Model it has to implement the Model interface. The Model interface covers two basic methods, which are needed by the simulator kernel. Additionally it acts as a marker interface for the manipulation subsystem to enable advanced operations on models. To make this class an *OpenSimKit* Model it is possible to add “implement Model” at the class definition. Another possibility is to derive the TemperatureSensor from the abstract base class BaseModel, which implements not only Model, but also lot's of convenience members and methods for the current *OpenSimKit* Model programming model.

```
public class TemperatureSensor extends BaseModel {
```

Currently a Model needs a set of specific code for interaction with the numerical solver. This includes a set of static member variables which can be different for each Model.

```
 /** Type of model. */
```

```
private static final String TYPE      = "TemperatureSensor";
/** Solver of model. */
private static final String SOLVER    = "none";
/** Maximum time step. */
private static final double MAXTSTEP  = 10.0;
/** Minimum time step. */
private static final double MINTSTEP  = 0.001;
/** Used time step. */
private static final int    TIMESTEP  = 1;
/** Regulstep. */
private static final int    REGULSTEP = 0;
```

After this part of necessary code follows the only member variable needed for the TemperatureSensor: the temperature. Choose double as its type and make it private.

```
/** The temperature this sensor measures. It is annotated with
 Readable so that the Tabgenerator can access it. */
private double temperature;
```

Additionally, the temperature should be accessible so it can be printed into the result file or inspected during run-time. However, it should not be pre-defined by the configuration file or manipulated during run-time. So it is annotated with @Readable.

```
/** The temperature this sensor measures. It is annotated with
 Readable so that the Tabgenerator can access it. */
@Readable private double temperature;
```

Now the ports are defined.

```
/* The input port of the temperature sensor. It accepts only data of the
 type PureGasDat. */
@IsPort private Port<PureGasDat> inputPort;
/* The output port of the temperature sensor. It accepts only data of the
 type PureGasDat. */
@IsPort private Port<PureGasDat> outputPort;
```

Next a constructor has to be written to allow for the correct instantiation of the TemperatureSensor. If just the default constructor is used this class cannot be used as a Model. As it is clearly visible, basically the parent class constructor is invoked with class-specific customizations.

```
public TemperatureSensor(final String name) {
  super(name, TYPE, SOLVER, MAXTSTEP, MINTSTEP, TIMESTEP, REGULSTEP);
}
```

Finally, the temperature reading method has to be implemented. For this the iterationStep() method of the base class is overridden.

```
@Override
public int iterationStep() throws IOException {
  /* Get the gas from the port. */
  PureGasDat indat = inputPort.readFromPort();
  /* Retrieve the temperature of the gas. */
  temperature = indat.temperature;
  /* Just write the gas liquid to the output. */
  outputPort.writeToPort(indat);

  return 0;
}
```

The complete listing of the TemperatureSensor Model (TemperatureSensor.java):

```
package org.opensimkit.models;

import java.io.IOException;
import org.opensimkit.BaseModel;
import org.opensimkit.ports.PureGasDat;
import org.opensimkit.manipulation.Readable;

/**
 * A simple temperature sensor model which is explained in the manual.
 */
public class TemperatureSensor extends BaseModel {
 /** Type of model. */
 private static final String TYPE      = "TemperatureSensor";
 /** Solver of model. */
 private static final String SOLVER    = "none";
 /** Maximum time step. */
 private static final double MAXTSTEP  = 10.0;
 /** Minimum time step. */
 private static final double MINTSTEP  = 0.001;
 /** Used time step. */
 private static final int    TIMESTEP  = 1;
 /** Regulstep. */
 private static final int    REGULSTEP = 0;
```

```
 /** The temperature this sensor measures. It is annotated with
Readable so that the Tabgenerator can access it. */
 @Readable private double temperature;


 /* The input port of the temperature sensor. It accepts only data of the
  type PureGasDat. */
 @IsPort private Port<PureGasDat> inputPort;
 /* The output port of the temperature sensor. It accepts only data of the
  type PureGasDat. */
 @IsPort private Port<PureGasDat> outputPort;


 /** Creates a new instance of TemperatureSensor.
  *
  * @param name Name of the temperature sensor.
  */
 public TemperatureSensor(final String name) {
   super(name, TYPE, SOLVER, MAXTSTEP, MINTSTEP, TIMESTEP, REGULSTEP);
 }


 @Override
 public int iterationStep() throws IOException {
   /* Get the gas from the port. */
   PureGasDat indat = inputPort.readFromPort();
   /* Retrieve the temperature of the gas. */
   temperature = indat.temperature;
   /* Just write the input gas to the output. */
   outputPort.writeToPort(indat);


   return 0;
 }
```

## 16.3 Compiling the Java Class

In current software releases it is necessary to compile the entire simulator to include the new model. The Ant build file has to be executed to build the whole simulator like described in the read me.

## 16.4 Modifying the Input File

To use the TemperatureSensor inside the simulation, the configuration file describing the

simulation has to be adapted. The configuration file includes the model definition, the mesh definition, the connections, and the logging definition.

To show the functionality of the TemperatureSensor it is not necessary to create a sample configuration file from scratch. It is easier to modify an existing configuration file. To do this a copy of the file Rocket-Stage-Simulation.xml with the name “Rocket-Stage-Simulation_sensor.xml”is created. At the end of the “models” section the lines below have to be appended. The fully qualified name of the TemperatureSensor class is “org.opensimkit.models.TemperatureSensor” and its name is chosen as “20_TemperatureSensor”. As the TemperatureSensor does not have any variable requiring initialisation, in other words no variable annotated by @Manipulatable, nothing needs to be written in between the “model” tags.

Rocket-Stage-Simulation.xml

```
<models>
.
.
.
  <!-- Begin changes for TemperatureSensor test. -->
    <model name="20_TemperatureSensor"
           class="org.opensimkit.models.TemperatureSensor">
    </model>
  <!-- End changes for TemperatureSensor test. -->
</models>
```

Inside the TankPressurization_sensor.xml more modifications are needed. First the “meshes” section needs to be modified. Inside the first mesh a reference to the Temperature Sensor model has to be added. This is done by putting the Model name in an element named “model”. The Temperature Sensor has the name “20_TemperatureSensor”. It is added at the end just after the Model with the name “19_FluidFlowValve”.

Rocket-Stage-Simulation_sensor.xml

```
<meshes>
  <mesh name="m0" level="top">
    <mesh>mesh_1</mesh>
    <model>05_Pipe</model>
    <model>06_Filter</model>
    <model>07_Pipe</model>
    <model>08_PReg</model>
    <model>09_Pipe</model>
    <model>10_Split</model>
    <model>11_Pipe</model>
    <model>12_PReg</model>
    <model>13_Pipe</model>
```

```
    <model>14_Pipe</model>
    <model>15_PReg</model>
    <model>16_Pipe</model>
    <model>17_Tank</model>
    <model>18_FluidFlowValve</model>
    <model>19_FluidFlowValve</model>
    <!-- Begin changes for TemperatureSensor test. -->
    <model>20_TemperatureSensor</model>
    <!-- End changes for TemperatureSensor test. -->
  </mesh>
  .
  .
  .
</meshes>
```

Next the connections have to be modified to allow for a proper inclusion of the Temperature Sensor. At the end the following XML block is inserted replacing the existing elements for the connection named “17_PureGas”. Basically the Temperature Sensor is inserted between the Models “16_Pipe” and “17_Tank”.

```
<connections>
  .
  .
  .
  <!-- Begin changes for TemperatureSensor test. -->
  <connection name="17_PureGas" class=”org.opensimkit.ports.PureGasPort”>
    <from model="16_Pipe"              port="outputPort"/>
    <to   model="20_TemperatureSensor" port="inputPort"/>
  </connection>
  <connection name="17a_PureGas" class=”org.opensimkit.ports.PureGasPort”>
    <from model="20_TemperatureSensor" port="outputPort"/>
    <to   model="17_Tank"              port="inputPort"/>
  </connection>
  <!-- End changes for TemperatureSensor test. →
</connections>
```

Finally the “logOutput” section has to be modified to print the TemperatureSensor's measured temperature into the output file. At the end of the “logOutputL section the following XML block is added. The attribute model must be the name of the Model, here “20_TemperatureSensor”. The attribute Variable needs the name of the variable to print into the output file. In this case it is the variable “temperature”. Note that the variable written in this place needs at least to be annotated with @Readable in the source of the Java class!

```
<!-- Begin changes for TemperatureSensor test. -->
<logOutput start="0.0" end="250.0" factor="10" delimeter="\t">
  .
  .
  .
  <entry model="20_TemperatureSensor" variable="temperature"/>
  <!-- End changes for TemperatureSensor test. -->
</logOutput>
```

## 16.5 Analyzing the Output File

If the output file (outfile.txt) is opened there should be an additional column named “S18-TEMPERATURE” in the file. If the default simulator settings were used the values of the output file for the S18-TEMPERATURE column should look like this:

```
S18-TEMPERATURE
300.000000
301.272749
302.792946
304.637163
306.531399
308.227183
309.544285
310.382221
310.713733
310.568240
310.011523
309.126704
307.999127
306.706321
305.312682
303.867866
302.407565
300.955543
299.526275
298.127368
296.761670
295.428949
294.127142
292.853235
291.603823
290.375467
289.164980
287.969420
286.786121
285.612714
284.447013
283.287346
282.132376
280.980871
279.831664
278.683607
277.535991
276.388402
275.240327
274.091226
272.940610
271.788361
270.634546
269.479107
268.321976
267.163104
266.002430
264.839882
263.675404
262.508907
261.340189
```

# 17 Coding Guidelines

During *OpenSimKit* porting/developing from C++ to Java, a few guidelines consistently have been applied. They are presented here and should be followed by all contributors:

- No wildcard imports.
  http://stackoverflow.com/questions/147454/why-is-using-a-wild-card-with-a-java-import-statement-bad/147532
- No static imports.
  http://www.javapractices.com/topic/TopicAction.do?Id=195
- Only one return.
- No singletons.
  http://code.google.com/p/google-singleton-detector/wiki/WhySingletonsAreControversial
  http://misko.hevery.com/2008/08/17/singletons-are-pathological-liars/
  http://www.xflex.cn/blog/archive/2009/03/why-singleton-is-bad-and-how-to-avoid.html
  http://blogs.msdn.com/scottdensmore/archive/2004/05/25/140827.aspx

# 18 Using the *OpenSimKit* Logger

## 18.1 Logger Code Entries for Model Classes

1. add slf4j imports
2. create one static logger per class, named after the class.
3. perform all output using the logger, do not use System.out.println !
4. use optimized syntax when dynamically assembling output messages (reasoning: string is not computed prior to logging call, but only if output is actually activated for the specific logger -> performance optimization)
5. Rules of writing a message: perform no formatting, do not add "Warning"/"Error" prefix or class/method information, only pure message text. Perform Formatting using the config file.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


public class MyClass {
  private static final Logger LOG = LoggerFactory.getLogger(MyClass.class);


  public void someMethod() {
    LOG.debug("a debugging message");
    LOG.info("some output for the user");
    LOG.warn("a warning message");
    LOG.error("an error message");
    LOG.error("an exception will be raised: ", new java.io.IOException());


    System.out.println("some output") // do not use this!
    LOG.debug("Entry number: " + i + " is " + String.valueOf(entry[i]));
    LOG.info("Entry number: {}", i);  // good practice for 1 argument
    LOG.warn("Entry number: {} is {}", i, String.valueOf(entry[i]));    // good
practice for 2 arguments
    LOG.error("Value {} was inserted between {} and {}.", new Object[] {newVal,
below, above});  // good practice for >2 arguments


[when nr of {} is higher than arguments, it is printed plainly, when nr of {}
is less than arguments, additional arguments are ignored]
}
```

## 18.2 Logger Configuration for Simulation Runs

If no custom configuration is provided, a default configuration is used:

General logback default:

- logging output to be directed on the console
- log level DEBUG

*OpenSimKit* default:

- level INFO
- INFO output to console (stdout) and logfile, located at execution path (usually osk-j-src\work)
- WARN output to console (stderr)
- ERROR output to console (stderr)

Logging properties (level, destination and format) can be configured on a per-class basis (inheritance through package hierarchy).

1. Create a logback xml configuration file

2.a) use the following option when executing the program: java -Dlogback.configurationFile=path-to-config-file (implemented in *OpenSimKit* .bat/.sh starting scripts)

2.b) name the file logback.xml or logback-test.xml and place it into the classpath

logback.xml should be used for release settings, logback-test.xml has priority over logback.xml and can be used during development.

```
<configuration>
  <root level="INFO">
    <appender-ref ref="STDOUT-INFO" />
    <appender-ref ref="STDOUT-DEBUG" />
    <appender-ref ref="STDOUT-DBGSHORT" />
    <appender-ref ref="STDERR-WARN" />
    <appender-ref ref="STDERR-ERR" />
    <appender-ref ref="ERRORLOG-FILE" />
  </root>


  <logger name="org.opensimkit.Kernel" level="DEBUG"/>
  <logger name="org.opensimkit.models.rocketpropulsion.FilterT1"
level="TRACE"/>
</configuration>
```

# 19 Literature


[1] Eickhoff, Jens:
Erstellung und Programmierung eines Rechenverfahrens zur thermodynamischen Erfassung des Druckgas-Fördersystems der Ariane L5 Stufe und Berechnung des nötigen Heliumbedarfs zur Treibstoffförderung,
Study Thesis, Institut für Thermodynamik der Luft- und Raumfahrt,
Universität Stuttgart, Pfaffenwaldring 31, Stuttgart, Germany 1988.

[2] Eickhoff, Jens:
Modulare Programmarchitektur für ein wissensbasiertes Simulationssystem mit erweiterter Anwendbarkeit in der Entwicklung und Betriebsüberwachung verfahrenstechnischer Anlagen,
Dissertation der TU Hamburg-Harburg,
VDI-Fortschrittsberichte, Reihe 20, Nr. 196, VDI-Verlag 1996, Germany
ISBN 3-18-319620-4

[3] Eickhoff, Jens:
Simulating Spacecraft Systems,
Springer-Verlag Berlin,
Series: Springer Aerospace Technology, Vol. 1,
ISBN: 978-3-642-01275-4

[4] http://opensource.gsfc.nasa.gov/projects/JAT/JAT.php
and
http://jat.sourceforge.net/

[5] G. Engeln-Muellges, F.Reutter
Formelsammlung zur numerischen Mathematik mit Standard-FORTRAN-77-Programmen", 5. Aufl.
B.I. Wissenschaftsverlag , Bibliographisches Institut
Mannheim/Wien/Zuerich, 1986

[6] Laurel, Chris.:
http://en.wikipedia.org/wiki/Celestia

[7] http://www.shatters.net/celestia/

[8] Adams, Douglas:
The Hitchhikers Guide to the Galaxy,
Pan Books, UK, 1979

[9] Ierusalimschy, Roberto:
Programming in Lua, Lua.org, December 2003
ISBN 85-903798-1-7

[10] http://www.lua.org/

[11] Nehab, Diego:
LuaSocket,
http://www.tecgraf.puc-rio.br/~diego/professional/luasocket/

[12] http://luaforge.net/projects/luasocket/

# Annexes

# 20 *OpenSimKit* History and Releases

| Release | Date | Changes |
|---|---|---|
| 0 | 1988 | Simulation of a rocket tank pressurization system in FORTRAN77 - see [01].<br>Jens Eickhoff |
| 1.0 | 1996 | ObjectSim 2.0.3 from [02] being the predecessor of *OpenSimKit*.<br>Jens Eickhoff |
| 2.0 | 2004 | First *OpenSimKit* C++ release.<br>Jens Eickhoff |
| 2.1 | Dec 2004 | C++ V2.1 cleaned up code release no longer comprising unused interfaces.<br>Jens Eickhoff |
| 2.2 | Feb. 2005 | Featuring XML inputfiles.<br>Peter Heinrich, ETH Zürich |
| 2.3 | Mar. 2005 | Featuring XML Inputfile editor.<br>Peter Heinrich, ETH Zürich |
| 2.4 | Aug. 2006 | Div. bug fixes & first Doxygen C++ code docu.<br>Jochen Scheikl, Matthias Raif, TU München |
| 2.4.1 | Sept. 2006 | "Simple Power" electric system demo.<br>Jens Eickhoff |
| 2.4.2 | Jul. 2007 | Java control console & simulator demo as multi program multi threading application.<br>Jens Eickhoff |
| OSK-J V2.4.2 | May 2008 | OSK-J 2.4 in Java - port from C++ to Java.<br>Alexander Brandt |
| OSK-J V2.4.4 | July 2008 | First version with full Ant-based build system.<br>Alexander Brandt |
| OSK-J V2.4.5 | Sept. 2008 | Port of MatLab SimplePower Model to Java.<br>Ivan Kossev, Universität Stuttgart |
| OSK-J V2.4.6 | Oct. 2008 | Provision of a simple Orbit Propagator in Java.<br>Ivan Kossev, Universität Stuttgart |
| OSK-J V2.5.0 | 01.11.2008 | Version featuring new XML File Structure, revised inputfile parsing and entirely new model/kernel registration mechanisms. First approach in direction of a service oriented architecture.<br>Alexander Brandt |
| OSK-J V2.5.1 | Feb. 2009 | Now implementing absolute simulation time in ISO format.<br>Alexander Brandt |

| Release | Date | Changes |
|---|---|---|
| OSK-J V2.6.0 | May 2009 | First release containing a first part of the results of the 1st *OpenSimKit* Developers Weekend.<br>Mario Kobald, Artur Bohr, Universität Stuttgart |
| OSK-J V2.6.5 | May 2009 | Introduced named ports and reworked the XML input file.<br>Alexander Brandt |
| OSK-J V2.6.7 | June 2009 | Introduced logging into the simulator and the model library.<br>Timm Pieper |
| OSK-J V2.7.0 | July 2009 | Release with fluid flow valves for engine fuel and oxidizer flow control as well as an engine controller steering these flow valves. Entire system thermodynamics revised to be robust against engine shutoff (flows = 0.0) and re-ignition.<br>Jens Eickhoff |
| OSK-J V3.0.0 | August 2009 | This release comprises essential results from the 1st *OpenSimKit* Developers Weekend. It completes the rocket simulation by a simplistic Structure Model and includes the realistic Earth Gravity model from the Java Astrodynamics Toolkit (JAT). This allows first orbit simulations around Earth including delta-V effects.<br><br>Mario Kobald, Artur Bohr, Claas Ziemke, Universität Stuttgart,<br>Jens Eickhoff |
| OSK-J V3.1.0 | September 2009 | Further results from the 1st *OpenSimKit* Developers Weekend have been added. This release comprises the plot visualization of simulation results.<br>Michael Fritz, Universität Stuttgart |
| OSK-J V3.2.0 | November 2009 | First results from the 2nd *OpenSimKit* Developers Weekend have been added:<br>Enhanced S/C dynamics model computing positions & velocities now both in Earth centered inertial frame coordinates, in Earth centered Earth fixed frame and as positions in longitude / latitude / altitude format.<br>A more realistic model of the rocket engine computing thrust according to fuel/oxidizer mixture and depending on flight altitude (atmospheric pressure).<br>Ivan Kossev, Mario Kobald, Fabian Steinmetz, Christoph Gomringer, Helmut Koch, Universität Stuttgart |

| Release | Date | Changes |
|---|---|---|
| OSK-J V3.3.0 | December 2009 | IntervalController model added which can be used as alternative to the EngineController.<br>By means of this model e.g. multiple engine firing / off time intervals can be defined.<br>The PointMass model was updated to interpret resulting forces as being always tangential to orbit flight vector until a detailed attitude dynamics is available.<br>A first use of both new PointMass and IntervalController was to simulate a first rocket stage Hohmann orbit transfer.<br><br>PointMass enhancements by Ivan Kossev, Universität Stuttgart,<br><br>IntervalController, Inputfile configuration for Hohmann Transfer and user manual enhancements Jens Eickhoff. |
| OSK-J V3.4.0 | December 2009 | This release comes with an enhanced MMI featuring a S/C ground track plot over Earth map to track S/C position.<br><br>User manual enhancements Michael Fritz and Jens Eickhoff |
| OSK-J V3.5.0 | April 2010 | Comprising simple spacecraft attitude dynamics and interface to 3D-Astrodynamics toolkit Celestia.<br><br>3D geometry model and raw Celestia Lua scripts by Rouven Witt.<br><br>Attitude dynamics implementation, Ivan Kossev.<br>Code merges, system integration, tests, debugging, packaging and user manual enhancements, Jens Eickhoff. |
| OSK-J V3.6.0 | December 2010 | V3.6.0- Dynamically updating plot windows - including ground track plot of the rocket over Earth.<br>Michael Fritz.<br><br>Fix of a problem in Celestia with scripted spacecraft attitude via ECI quaternions.<br>Jens Eickhoff and Ivan Kossev.<br><br>Fix of longitude position computation formally implemented via Java Astrodynamics Toolkit.<br>Michael Fritz, Oliver Zeile, Jens Eickhoff.<br><br>User Manual updates.<br>Rouven Witt, Jens Eickhoff. |

| Release | Date | Changes |
|---|---|---|
| OSK-J<br>V3.7.0 | January 2011 | Implementation of provider/subscriber mechanism for model variable interchange.<br>Alexander Brandt, Jens Eickhoff<br><br>User Manual updates.<br>Michael Fritz, Ivan Kossev, Jens Eickhoff |

# 21 License, Trademark and Warranty Disclaimer

OpenSimKit®

Is a registered trademark of the original tool inventor and open source project coordinator,

Dr. Jens Eickhoff
Immenstaad,
Germany

*OpenSimKit* is free software. You are free to publish, copy, use and modify it under the conditions of the GNU General Public License Version 3 as published by the Free Software Foundation.

A copy of the GNU General Public License is enclosed in the tool distribution downloadable from

`www.opensimkit.org`

For all code from other libraries included in *OpenSimKit,* like NASA's Java Astrodynamics Toolkit, also the GPL applies with one exception:
Due to inclusion of the Woodstox Stax parser the following license information is to be considered in addition. This product includes software licensed under the Apache License Version 2.0 (http://www.apache.org/licenses/LICENSE-2.0).

Publishing and use of this software program is done under the assumption that it will be beneficial for the users, but WITHOUT ANY WARRANTY, even without any implicit warranty on READYNESS FOR MARKETING or the APPLICABILITY FOR A SPECIFIC PURPOSE. Details can be found in the GNU General Public License.